# Linux驱动\_按键与中断

## 1.ARM架构的Linux中断体系

### 1.1 一些常用的知识

#### 1.1.1 中断向量表位置

在ARM V4及V4T以后的大部分处理器中，中断向量表的基地址可以有两个位置：一个是0，另一个是0xffff0000。可以通过CP15协处理器c1寄存器中V 位(bit[13])控制。V和中断向量表的对应关系如下：

　　V**=**0 ～ 0x00000000**~**0x0000001C

　　V**=**1 ～ 0xffff0000**~**0xffff001C

#### 1.1.2 中断模式的stack准备

ARM处理器有多种processor mode，例如user mode（用户空间的AP所处于的模式）、supervisor mode（即SVC mode，大部分的内核态代码都处于这种mode）、IRQ mode（发生中断后，处理器会切入到该mode）等。对于linux kernel，其中断处理处理过程中，ARM 处理器大部分都是处于SVC mode。但是，实际上产生中断的时候，ARM处理器实际上是进入IRQ mode，因此在进入真正的IRQ异常处理之前会有一小段IRQ mode的操作，之后会进入SVC mode进行真正的IRQ异常处理。由于IRQ mode只是一个过度，因此IRQ mode的栈很小，只有12个字节，具体如下：

struct stack **{**

    u32 irq**[**3**];**

    u32 abt**[**3**];**

    u32 und**[**3**];**

**}** \_\_\_\_cacheline\_aligned**;**

static struct stack stacks**[**NR\_CPUS**];**

#### 1.1.3 SVC模式的stack准备

我们经常说进程的用户空间和内核空间，对于一个应用程序而言，可以运行在用户空间，也可以通过系统调用进入内核空间。在用户空间，使用的是用户栈，也就是我们软件工程师编写用户空间程序的时候，保存局部变量的stack。陷入内核后，当然不能用用户栈了，这时候就需要使用到内核栈。所谓内核栈其实就是处于SVC mode时候使用的栈。在linux最开始启动的时候，系统只有一个进程（更准确的说是kernel thread），就是PID等于0的那个进程，叫做swapper进程（或者叫做idle进程）。该进程的内核栈是静态定义的，如下：

union thread\_union init\_thread\_union \_\_init\_task\_data **=**

**{** INIT\_THREAD\_INFO**(**init\_task**)** **};**

union thread\_union **{**

    struct thread\_info thread\_info**;**

    unsigned long stack**[**THREAD\_SIZE**/sizeof(**long**)];**

对于ARM平台，THREAD\_SIZE是8192个byte，因此占据两个page frame。随着初始化的进行，Linux kernel会创建若干的内核线程，而在进入用户空间后，user space的进程也会创建进程或者线程。Linux kernel在创建进程（包括用户进程和内核线程）的时候都会分配一个（或者两个，和配置相关）page frame，具体代码如下：

static struct task\_struct **\***dup\_task\_struct**(**struct task\_struct **\***orig**)**

**{**

**......**

    ti **=** alloc\_thread\_info\_node**(**tsk**,** node**);**

**if** **(!**ti**)**

**goto** free\_tsk**;**

**......**

**}**

底部是struct thread\_info数据结构，顶部（高地址）就是该进程的内核栈。当进程切换的时候，整个硬件和软件的上下文都会进行切换，这里就包括了svc mode的sp寄存器的值被切换到调度算法选定的新的进程的内核栈上来。

### 1.2 中断的入口函数

#### 1.2.1 中断向量表

**.**equ stubs\_offset**,** \_\_vectors\_start **+** 0x200 **-** \_\_stubs\_start

**.**globl \_\_vectors\_start

\_\_vectors\_start**:**

swi SYS\_ERROR0 //系统复位中断

b vector\_und **+** stubs\_offset //未定义指令

ldr pc**,** **.**LCvswi **+** stubs\_offset //swi中断

b vector\_pabt **+** stubs\_offset //指令预取终止

b vector\_dabt **+** stubs\_offset //数据访问终止

b vector\_addrexcptn **+** stubs\_offset //保留

b vector\_irq **+** stubs\_offset //irq中断

b vector\_fiq **+** stubs\_offset //fiq中断

**.**globl \_\_vectors\_end

\_\_vectors\_end

这里以irq中断为例介绍，当发生irq中断时，硬件会帮忙自动执行b vector\_irq + stubs\_offset跳转到irq处理函数。  
在了解irq处理函数之前，先来分析一下中断向量表的搬移函数。

#define CONFIG\_VECTORS\_BASE 0xffff0000

void \_\_init early\_trap\_init**(**void**)**

**{**

unsigned long vectors **=** CONFIG\_VECTORS\_BASE**;**

extern char \_\_stubs\_start**[],** \_\_stubs\_end**[];**

extern char \_\_vectors\_start**[],** \_\_vectors\_end**[];**

extern char \_\_kuser\_helper\_start**[],** \_\_kuser\_helper\_end**[];**

int kuser\_sz **=** \_\_kuser\_helper\_end **-** \_\_kuser\_helper\_start**;**

memcpy**((**void **\*)**vectors**,** \_\_vectors\_start**,** \_\_vectors\_end **-** \_\_vectors\_start**);**

memcpy**((**void **\*)**vectors **+** 0x200**,** \_\_stubs\_start**,** \_\_stubs\_end **-** \_\_stubs\_start**);**

memcpy**((**void **\*)**vectors **+** 0x1000 **-** kuser\_sz**,** \_\_kuser\_helper\_start**,** kuser\_sz**);**

memcpy**((**void **\*)**KERN\_SIGRETURN\_CODE**,** sigreturn\_codes**,**

**sizeof(**sigreturn\_codes**));**

flush\_icache\_range**(**vectors**,** vectors **+** PAGE\_SIZE**);**

modify\_domain**(**DOMAIN\_USER**,** DOMAIN\_CLIENT**);**

**}**

该函数将\_\_vectors\_start到\_\_vectors\_end之间的代码拷贝到0xffff0000，  
 \_\_stubs\_start到\_\_stubs\_end之间的代码拷贝到0xffff0000+512。

#### 1.2.2 中断处理函数

##### 1.2.2.1 中断处理函数代码

还是以irq中断为例来分析，上面的b vector\_irq + stubs\_offset会跳转到vector\_stub irq, IRQ\_MODE, 4处。\_\_stubs\_end 至 \_\_stubs\_start之间是异常处理的位置，vector\_und、vector\_pabt、vector\_irq、vector\_fiq都在它们中间。

**.**globl \_\_stubs\_start

\_\_stubs\_start**:**

vector\_stub irq**,** IRQ\_MODE**,** 4

**.**long \_\_irq\_usr @ 0 **(**USR\_26 **/** USR\_32**)**

**.**long \_\_irq\_invalid @ 1 **(**FIQ\_26 **/** FIQ\_32**)**

**.**long \_\_irq\_invalid @ 2 **(**IRQ\_26 **/** IRQ\_32**)**

**.**long \_\_irq\_svc @ 3 **(**SVC\_26 **/** SVC\_32**)**

**.**long \_\_irq\_invalid @ 4

**.**long \_\_irq\_invalid @ 5

**.**long \_\_irq\_invalid @ 6

**.**long \_\_irq\_invalid @ 7

**.**long \_\_irq\_invalid @ 8

**.**long \_\_irq\_invalid @ 9

**.**long \_\_irq\_invalid @ a

**.**long \_\_irq\_invalid @ b

**.**long \_\_irq\_invalid @ c

**.**long \_\_irq\_invalid @ d

**.**long \_\_irq\_invalid @ e

**.**long \_\_irq\_invalid @ f

**...**

**.**globl \_\_stubs\_end

\_\_stubs\_end**:**

##### 1.2.2.2 stubs\_offset的作用

由于中断向量表和中断处理函数进行了移动，所以不能直接用b vector\_irq跳转到中断处理函数，需要计算出搬移后两个命令之间的偏移量，才能正确的进行跳转。搬移后vector\_irq符号的地址：

CONFIG\_VECTORS\_BASE **+** 0x200 **+** vector\_irq**(**编译地址**)** **-** \_\_stubs\_start**(**编译地址**)**

执行中断异常时，指令地址应该为：

CONFIG\_VECTORS\_BASE **+** （irq\_PC **-** \_\_vectors\_start）

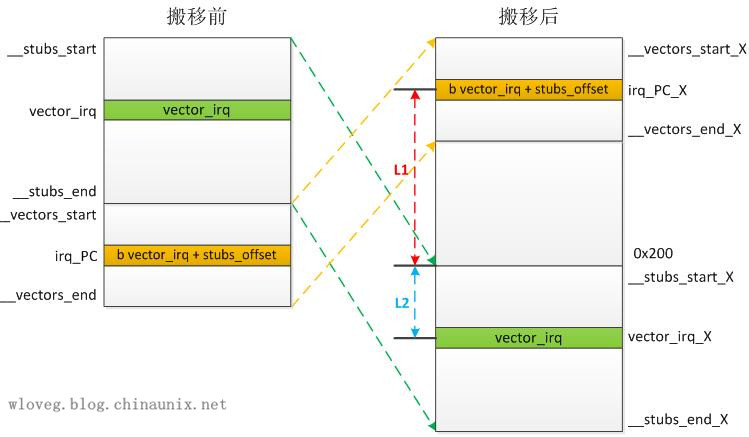
所以两条指令之间搬移后的偏移地址为：

（CONFIG\_VECTORS\_BASE **+** 0x200 **+** （vector\_irq**(**编译地址**)** **-** \_\_stubs\_start**(**编译地址**)**）） **-** （CONFIG\_VECTORS\_BASE **+** （irq\_PC**)** **–**

\_\_vectors\_start（编译地址）））

**=** （0x200 **+** （vector\_irq**(**编译地址**)** **-** \_\_stubs\_start**(**编译地址**)**）） **-** （irq\_PC **-** \_\_vectors\_start（编译地址））

**=** \_\_vectors\_start **+** 0x200 **-** \_\_stubs\_start **+** vector\_irq **-** irq\_PC



令stubs\_offset = \_\_vectors\_start + 0x200 - \_\_stubs\_start,则offset = vector\_irq + stubs\_offset -irq\_PC，所以中断入口点为  
b vector\_irq + stubs\_offset，其中减去irq\_PC是由汇编器在编译时完成的。

##### 1.2.2.3 vector\_irq

切入正题，我们目前并未发现关于vector\_irq的定义，vector\_stub irq, IRQ\_MODE, 4语句展开后就是vector\_irq，相关宏定义如下：

**.**macro vector\_stub**,** name**,** mode**,** correction**=**0

**.**align 5

vector\_\name**:**

//当发生IRQ中断的时候，lr中保存了发生中断的PC＋4，

//如果减去4的话，得到的就是发生中断那一点的PC值。

**.if** \correction

sub lr**,** lr**,** #\correction

**.**endif

//当前是IRQ mode,SP\_irq在初始化的时候已经设定(12个字节)

//在irq mode的stack上，依次保存了发生中断那一点的r0值、PC值以及CPSR值

//(具体操作是通过spsr进行的，其实硬件已经帮我们保存了CPSR到SPSR中了)

//为何要保存r0值？因为随后的代码要使用r0寄存器，

//因此我们要把r0放到栈上，只有这样才能完完全全恢复硬件现场。

stmia sp**,** **{**r0**,** lr**}** @ save r0**,** lr

mrs lr**,** spsr

str lr**,** **[**sp**,** #8**]** @ save spsr

//可怜的IRQ mode稍纵即逝，这段代码就是准备将ARM推送到SVC mode

//如何准备？其实就是修改SPSR的值，SPSR不是CPSR，

//不会引起processor mode的切换（毕竟这一步只是准备而已）。

mrs r0**,** cpsr

eor r0**,** r0**,** #**(**\mode **^** SVC\_MODE**)**

msr spsr\_cxsf**,** r0

//很多异常处理的代码返回的时候都是使用了stack相关的操作，这里没有

//“movs    pc, lr ”指令除了字面上意思（把lr的值付给pc），

//还有一个隐含的操作（movs中‘s’的含义）：把SPSR copy到CPSR，从而实现了模式的切换。

**and** lr**,** lr**,** #0x0f  //获取被中断前，处理器所处的模式

mov r0**,** sp //让r0寄存器指向中断模式下堆栈的基地址

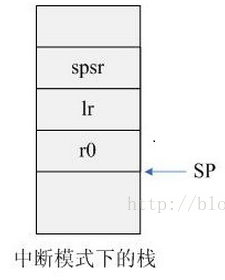
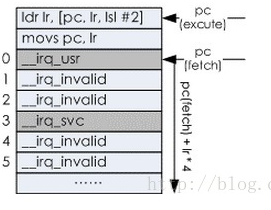
ldr lr**,** **[**pc**,** lr**,** lsl #2**]** //获取跳转的位置

movs pc**,** lr @ branch to handler in SVC mode

**.**endm

当发生IRQ中断时，CPU自动完成的工作有：  
(1)IRQ模式下的R14(LR)寄存器保存当前PC值+4或者+8。  
(2)将CPSR的值复制到IRQ模式的SPSR寄存器。  
(3)将CPSR的工作模式位设置为IRQ模式  
(4)PC值跳到中断向量表IRQ的位置。

在vector\_irq中，主要完成如下工作(切换到SVC模式)：  
(1)IRQ的栈保存了spsr,lr,r0寄存器值，也就是保存了发生中断时的CPSR,PC,r0寄存器的值。  
(2)修改SPSR，准备从CPSR切换到SVC模式。  
(3)获取发生中断前，处理器是处于USER还是SVC模式，根据处理器模式的不同，分别跳转到\_\_irq\_usr和\_\_irq\_svc两个分支。并将r0寄存器指向IRQ模式下的SP寄存器。

寄存器结构体定义如下：

struct pt\_regs **{**

long uregs**[**18**];**

**};**

#define ARM\_cpsr uregs[16]

#define ARM\_pc uregs[15]

#define ARM\_lr uregs[14]

#define ARM\_sp uregs[13]

#define ARM\_ip uregs[12]

#define ARM\_fp uregs[11]

#define ARM\_r10 uregs[10]

#define ARM\_r9 uregs[9]

#define ARM\_r8 uregs[8]

#define ARM\_r7 uregs[7]

#define ARM\_r6 uregs[6]

#define ARM\_r5 uregs[5]

#define ARM\_r4 uregs[4]

#define ARM\_r3 uregs[3]

#define ARM\_r2 uregs[2]

#define ARM\_r1 uregs[1]

#define ARM\_r0 uregs[0]

#define ARM\_ORIG\_r0 uregs[17]

##### 1.2.2.4 \_\_irq\_usr

当发生中断的时候，代码运行在用户空间,则会运行\_\_irq\_usr。

**.**align 5

\_\_irq\_usr**:**

usr\_entry

get\_thread\_info tsk

irq\_handler

mov why**,** #0

b ret\_to\_user

(1)usr\_entry 保存发生中断时候的现场。所谓保存现场其实就是把发生中断那一刻的硬件上下文（各个寄存器）保存在了SVC mode的stack上。

include**/**asm**-**arm**/**asm**-**offsets**.**h**:**40**:**#define S\_PC 60 /\* offsetof(struct pt\_regs, ARM\_pc) @ \*/

**.**macro usr\_entry

//进入svc模式，sp切换到svc模式的sp

//S\_FRAME\_SIZE为18\*4=72，也就是需要保存18个寄存器

sub sp**,** sp**,** #S\_FRAME\_SIZE

//svc栈压入r1～r12

//stmib中的ib表示increment before，因此，在压入R1的时候，stack pointer会先增加4，重要是预留r0的位置

//stmib没有！的修饰符，表示压栈完成后并不会真正更新stack pointer，因此sp保持原来的值

stmib sp**,** **{**r1 **-** r12**}**

//这里r0指向了irq stack

//因此，r1是中断时候的r0值，r2是中断现场的PC值，r3是中断现场的CPSR值

ldmia r0**,** **{**r1 **-** r3**}**

//r0 = sp+60,将r0指向ARM\_pc

add r0**,** sp**,** #S\_PC

//r4保存-1

mov r4**,** #**-**1

//保存中断那一刻的r0 ,保存在ARM\_r0

str r1**,** **[**sp**]**

//将r2 r2 r3依次保存到ARM\_pc ARM\_cpsr ARM\_ORIG\_r0

stmia r0**,** **{**r2 **-** r4**}**

//我们保存的是发生中断那一刻（对于本节，这是当时user mode的sp和lr），指令中的^符号表示访问user mode的寄存器。

stmdb r0**,** **{**sp**,** lr**}^**

//alignment\_trap在配置CONFIG\_ALIGNMENT\_TRAP时有效，如果开启了该选项，中断处理中将支持对齐跟踪

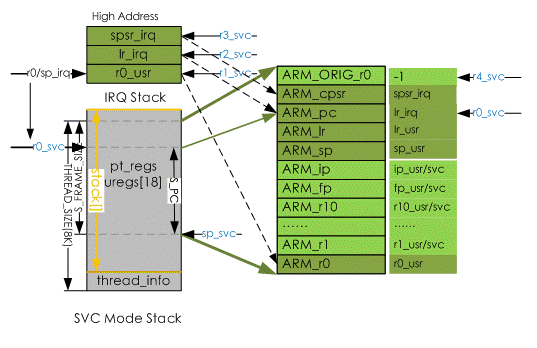
alignment\_trap r0

//zero\_fp用来设置fp栈帧寄存器为0

zero\_fp

**.**endm

usr\_entry函数操作过程示意如下图所示：



(2)get\_thread\_info

根据当前sp指针，将该指针最右边13位清0，获得当前任务的thread\_info。

**.**macro get\_thread\_info**,** rd

mov \rd**,** sp**,** lsr #13

mov \rd**,** \rd**,** lsl #13

**.**endm

其中，进程栈的栈底，存储着该进程的基本信息如id 调度优先级等信息。  
栈顶指针低13位清0，显然高位是base地址，低13位看来是整个进程栈的偏移地址，由此可以得到栈的base地址。

tsk .req r9 @ current thread\_info

thread\_info和栈的关系。

union thread\_union **{**

    struct thread\_info thread\_info**;**  /\*线程属性\*/

    unsigned long stack**[**THREAD\_SIZE**/sizeof(**long**)];**  /\*栈\*/

**};**

(3)irq\_handler

中断服务例程，核心处理内容，后面再进行分析。

(4) mov why**,** #0

r8寄存器赋值为0。

why .req r8 @ Linux syscall **(!=** 0**)**

(5) b ret\_to\_user

返回到用户模式，中断之前的状态。

ENTRY**(**ret\_to\_user**)**

ret\_slow\_syscall**:**

disable\_irq @ disable interrupts

//从任务的TI\_FLAGS标志判断是否需要处理抢占或者信号。

ldr r1**,** **[**tsk**,** #TI\_FLAGS**]**

tst r1**,** #\_TIF\_WORK\_MASK

//处理抢占或者信号

bne work\_pending

//没有抢占或者信号需要处理，或者已经处理完毕，开始退回用户态

no\_work\_pending**:**

//在返回用户态前，处理各个体系结构的钩子

arch\_ret\_to\_user r1**,** lr

//寄存器恢复,回到中断之前的状态

//获取中断时的cpsr寄存器值

ldr r1**,** **[**sp**,** #S\_PSR**]** @ get calling cpsr

//获取中断时的pc值,这里会更新sp的值，sp = sp+60

ldr lr**,** **[**sp**,** #S\_PC**]!** @ get pc

//将发生中断时的cpsr的值赋给spsr\_svc

msr spsr\_cxsf**,** r1 @ save in spsr\_svc

//恢复r0-lr寄存器值，并将spsr的值赋cpsr

ldmdb sp**,** **{**r0 **-** lr**}^** @ get calling r1 **-** lr

//nop 不做任何事情

mov r0**,** r0

//恢复内核栈到中断产生之前的位置。

//sp = sp+72-60 = sp+12

//目前sp的位置为sp+60

add sp**,** sp**,** #S\_FRAME\_SIZE **-** S\_PC

//返回中断前的代码处

movs pc**,** lr @ **return** **&** move spsr\_svc into cpsr

##### 1.2.2.5 \_\_irq\_svc

当发生中断的时候，代码运行在内核空间,则会运行\_\_irq\_svc。

**.**align 5

\_\_irq\_svc**:**

svc\_entry

irq\_handler

ldr r0**,** **[**sp**,** #S\_PSR**]** @ irqs are already disabled

msr spsr\_cxsf**,** r0

ldmia sp**,** **{**r0 **-** pc**}^** @ load r0 **-** pc**,** cpsr

(1)svc\_entry

保存发生中断那一刻的现场保存在内核栈上，和usr模式处理中断不太一样的是，usr模式发生中断时，cpu模式变化为usr --> irq --> svc，而svc模式发生中断后，cpu模式变化为svc --> irq --> svc。

include**/**asm**-**arm**/**asm**-**offsets**.**h**:**38**:**#define S\_SP 52 /\* offsetof(struct pt\_regs, ARM\_sp) @ \*/

**.**macro svc\_entry

//分配栈空间保存18个寄存器

sub sp**,** sp**,** #S\_FRAME\_SIZE

//sp先增加4，保存r1-r12到对应的栈中

stmib sp**,** **{**r1 **-** r12**}**

//恢复r0寄存器的内容到r1-r3

//r1是中断时候的r0值，r2是中断现场的PC值，r3是中断现场的CPSR值

ldmia r0**,** **{**r1 **-** r3**}**

//r5 = sp+52 ARM\_sp寄存器位置

add r5**,** sp**,** #S\_SP

//r4 = -1

mov r4**,** #**-**1

//r0 = sp+72

//发生中断时sp指针的位置

//因为是从svc模式进入IRQ，再回到svc模式，所以发生中断时的sp指针就为sp+72

add r0**,** sp**,** #S\_FRAME\_SIZE

//保存真正的r0到sp

str r1**,** **[**sp**]**

//将lr寄存器的值暂存在r1

mov r1**,** lr

//保存中断时的 SP PC CPSR 值到栈中

stmia r5**,** **{**r0 **-** r4**}**

**.**endm

(2) irq\_handler

中断处理函数，在后面进行分析。

(3) 剩余部分

将spsr\_cxsf的内容读到栈的ARM\_CPSR位置处，最后恢复所有的寄存器。

ldr r0**,** **[**sp**,** #S\_PSR**]** @ irqs are already disabled

msr spsr\_cxsf**,** r0

ldmia sp**,** **{**r0 **-** pc**}^** @ load r0 **-** pc**,** cpsr

include**/**asm**-**arm**/**asm**-**offsets**.**h**:**41**:**#define S\_PSR 64 /\* offsetof(struct pt\_regs, ARM\_cpsr) @ \*/

##### 1.2.2.6 irq\_handler

获取中断号，并跳转到C语言写的中断处理函数中。

**.**macro irq\_handler

//这里为空函数

get\_irqnr\_preamble r5**,** lr

//获取本次中断的硬件中断号,保存在r0中

1**:** get\_irqnr\_and\_base r0**,** r6**,** r5**,** lr

//如果上面函数设置了Z flag，则r1=sp，

//如果上面找到了中断号，Z flag = 1

movne r1**,** sp

//如果Z flag被设置了，则设置lr为标号1

//也就是asm\_do\_IRQ的返回地址

adrne lr**,** 1b

//如果Z flag被设置了，则跳到C语言的中断处理函数

//r0=中断号 r1=struct pt\_regs

bne asm\_do\_IRQ

**.**endm

获取中断号的代码如下：

**.**macro get\_irqnr\_and\_base**,** irqnr**,** irqstat**,** base**,** tmp

//对应 r0, r6, r5, lr

//mov r5, #S3C24XX\_VA\_IRQ

mov \base**,** #S3C24XX\_VA\_IRQ

//获取INTPND寄存器的数据 存入 r6

ldr \irqstat**,** **[** \base**,** #INTPND **]**

//测试r6的值，看是否为0

teq \irqstat**,** #0

//如果为0 就说明没有中断发生,INTPND寄存器中没有置1的位跳转到1002标号处

beq 1002f

//如果不为0，说明INTPND有位置1，这里读取INTOFFSET寄存器的内容到r0中

ldr \irqnr**,** **[** \base**,** #INTOFFSET **]**

//给lr赋值1

mov \tmp**,** #1

//测试r6和1<<r0 进行逻辑按位与操作

tst \irqstat**,** \tmp**,** lsl \irqnr

//如果上面为相等，那么就说明找到了中断号，就直接调到1001标号

bne 1001f

//如果不等，则继续执行,要自己计算中断号

//r0 = 0

mov \irqnr**,** #0

movs \tmp**,** \irqstat**,** lsl#16

addeq \irqnr**,** \irqnr**,** #16

moveq \irqstat**,** \irqstat**,** lsr#16

tst \irqstat**,** #0xff

addeq \irqnr**,** \irqnr**,** #8

moveq \irqstat**,** \irqstat**,** lsr#8

tst \irqstat**,** #0xf

addeq \irqnr**,** \irqnr**,** #4

moveq \irqstat**,** \irqstat**,** lsr#4

tst \irqstat**,** #0x3

addeq \irqnr**,** \irqnr**,** #2

moveq \irqstat**,** \irqstat**,** lsr#2

tst \irqstat**,** #0x1

addeq \irqnr**,** \irqnr**,** #1

1001**:**

//这里r0 ＝ r0 + IRQ\_EINT0，这里便得到了中断号

adds \irqnr**,** \irqnr**,** #IRQ\_EINT0

1002**:**

@@ exit here**,** Z flag unset **if** IRQ

**.**endm

##### 1.2.2.7 asm\_do\_IRQ

在irq\_desc结构体中找到对应的处理函数，并执行。后面则是需要分析irq\_desc结构体是如何被初始化的。

asmlinkage void \_\_exception asm\_do\_IRQ**(**unsigned int irq**,** struct pt\_regs **\***regs**)**

**{**

//记录中断现场

struct pt\_regs **\***old\_regs **=** set\_irq\_regs**(**regs**);**

struct irq\_desc **\***desc **=** irq\_desc **+** irq**;**

**if** **(**irq **>=** NR\_IRQS**)**

desc **=** **&**bad\_irq\_desc**;**

//irq\_enter就是告诉系统,现在正在处理中断的上半部分工作,不可以进行调度

irq\_enter**();**

//调用相应的处理函数

desc\_handle\_irq**(**irq**,** desc**);**

//AT91使用

irq\_finish**(**irq**);**

irq\_exit**();**

//恢复中断现场

set\_irq\_regs**(**old\_regs**);**

**}**

### 1.3 中断处理函数的初始化

start\_kernel会调用init\_IRQ()开始开始中断处理函数的初始化。

void \_\_init init\_IRQ**(**void**)**

**{**

int irq**;**

**for** **(**irq **=** 0**;** irq **<** NR\_IRQS**;** irq**++)**

irq\_desc**[**irq**].**status **|=** IRQ\_NOREQUEST **|** IRQ\_NOPROBE**;**

init\_arch\_irq**();**

**}**

其中init\_arch\_irq为s3c24xx\_init\_irq。  
s3c24xx\_init\_irq主要用于清空相关的中断寄存器，并设置中断的操作函数和处理函数。具体是如何调用这些初始化的函数进行中断响应，后面会解释。

void \_\_init s3c24xx\_init\_irq**(**void**)**

**{**

unsigned long pend**;**

unsigned long last**;**

int irqno**;**

int i**;**

//1.清空相应的寄存器

//清空EINTPEND寄存器

last **=** 0**;**

**for** **(**i **=** 0**;** i **<** 4**;** i**++)** **{**

pend **=** \_\_raw\_readl**(**S3C24XX\_EINTPEND**);**

**if** **(**pend **==** 0 **||** pend **==** last**)**

**break;**

\_\_raw\_writel**(**pend**,** S3C24XX\_EINTPEND**);**

printk**(**"irq: clearing pending ext status %08x\n"**,** **(**int**)**pend**);**

last **=** pend**;**

**}**

//清空INTPND寄存器

last **=** 0**;**

**for** **(**i **=** 0**;** i **<** 4**;** i**++)** **{**

pend **=** \_\_raw\_readl**(**S3C2410\_INTPND**);**

**if** **(**pend **==** 0 **||** pend **==** last**)**

**break;**

\_\_raw\_writel**(**pend**,** S3C2410\_SRCPND**);**

\_\_raw\_writel**(**pend**,** S3C2410\_INTPND**);**

printk**(**"irq: clearing pending status %08x\n"**,** **(**int**)**pend**);**

last **=** pend**;**

**}**

//清空SUBSRCPND寄存器

last **=** 0**;**

**for** **(**i **=** 0**;** i **<** 4**;** i**++)** **{**

pend **=** \_\_raw\_readl**(**S3C2410\_SUBSRCPND**);**

**if** **(**pend **==** 0 **||** pend **==** last**)**

**break;**

printk**(**"irq: clearing subpending status %08x\n"**,** **(**int**)**pend**);**

\_\_raw\_writel**(**pend**,** S3C2410\_SUBSRCPND**);**

last **=** pend**;**

**}**

//2.设置中断处理函数

//handle\_level\_irq 用于电平触发中断的流控处理；

//handle\_edge\_irq 用于边沿触发中断的流控处理；

//set\_irq\_chip 用于设置这些外部中断的触发方式（电平触发，边沿触发），使能中断，禁止中断

//set\_irq\_handler 设置中断处理函数

//set\_irq\_chained\_handler 设置串行中断处理函数，也就是一个中断号中还有其他子中断号，需要进一步处理。设置串行时，会开启该中断源，这样下面的子中断才会得到处理

**for** **(**irqno **=** IRQ\_EINT4t7**;** irqno **<=** IRQ\_ADCPARENT**;** irqno**++)** **{**

**switch** **(**irqno**)** **{**

**case** IRQ\_EINT4t7**:**

**case** IRQ\_EINT8t23**:**

**case** IRQ\_UART0**:**

**case** IRQ\_UART1**:**

**case** IRQ\_UART2**:**

**case** IRQ\_ADCPARENT**:**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_level\_chip**);**

set\_irq\_handler**(**irqno**,** handle\_level\_irq**);**

**break;**

**case** IRQ\_RESERVED6**:**

**case** IRQ\_RESERVED24**:**

/\* no IRQ here \*/

**break;**

**default:**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_chip**);**

set\_irq\_handler**(**irqno**,** handle\_edge\_irq**);**

//该函数主要是为特定的中断设置相应的状态标记，

//而这里我们调用它的目的就是清掉IRQ\_NOREQUEST标记，

//告诉系统该中断已经可以被申请使用了，中断在申请的时候会查看是否有IRQ\_NOREQUEST标记，如有则表明该中断还不能使用。而初始化的时候所有的中断都有这个标记。

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

**}**

/\* setup the cascade irq handlers \*/

set\_irq\_chained\_handler**(**IRQ\_EINT4t7**,** s3c\_irq\_demux\_extint4t7**);**

set\_irq\_chained\_handler**(**IRQ\_EINT8t23**,** s3c\_irq\_demux\_extint8**);**

set\_irq\_chained\_handler**(**IRQ\_UART0**,** s3c\_irq\_demux\_uart0**);**

set\_irq\_chained\_handler**(**IRQ\_UART1**,** s3c\_irq\_demux\_uart1**);**

set\_irq\_chained\_handler**(**IRQ\_UART2**,** s3c\_irq\_demux\_uart2**);**

set\_irq\_chained\_handler**(**IRQ\_ADCPARENT**,** s3c\_irq\_demux\_adc**);**

/\* external interrupts \*/

**for** **(**irqno **=** IRQ\_EINT0**;** irqno **<=** IRQ\_EINT3**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (ext int)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_eint0t4**);**

set\_irq\_handler**(**irqno**,** handle\_edge\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

**for** **(**irqno **=** IRQ\_EINT4**;** irqno **<=** IRQ\_EINT23**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (extended s3c irq)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irqext\_chip**);**

set\_irq\_handler**(**irqno**,** handle\_edge\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

/\* register the uart interrupts \*/

irqdbf**(**"s3c2410: registering external interrupts\n"**);**

**for** **(**irqno **=** IRQ\_S3CUART\_RX0**;** irqno **<=** IRQ\_S3CUART\_ERR0**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (s3c uart0 irq)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_uart0**);**

set\_irq\_handler**(**irqno**,** handle\_level\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

**for** **(**irqno **=** IRQ\_S3CUART\_RX1**;** irqno **<=** IRQ\_S3CUART\_ERR1**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (s3c uart1 irq)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_uart1**);**

set\_irq\_handler**(**irqno**,** handle\_level\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

**for** **(**irqno **=** IRQ\_S3CUART\_RX2**;** irqno **<=** IRQ\_S3CUART\_ERR2**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (s3c uart2 irq)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_uart2**);**

set\_irq\_handler**(**irqno**,** handle\_level\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

**for** **(**irqno **=** IRQ\_TC**;** irqno **<=** IRQ\_ADC**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (s3c adc irq)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_adc**);**

set\_irq\_handler**(**irqno**,** handle\_edge\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

irqdbf**(**"s3c2410: registered interrupt handlers\n"**);**

**}**

### 1.4 中断注册

在驱动程序需要调用request\_irq为irq注册实际的中断处理函数。例如：

request\_irq**(**IRQ\_EINT0**,** buttons\_irq**,** IRQT\_BOTHEDGE**,** "S2"**,** **&**pins\_desc**[**0**]);**

函数原型：

int request\_irq**(**unsigned int irq**,** irq\_handler\_t handler**,**unsigned long irqflags**,** const char **\***devname**,** void **\***dev\_id**)**

**irq**是要申请的硬件中断号。  
**handler**是向系统注册的中断处理函数，是一个回调函数，中断发生时，系统调用这个函数，dev\_id参数将被传递给它。  
**irqflags**是中断处理的属性，若设置了IRQF\_DISABLED，则表示中断处理程序是快速处理程序，快速处理程序被调用时屏蔽所有中断，慢速处理程序不屏蔽；若设置了IRQF\_SHARED，则表示多个设备共享中断，若设置了IRQF\_SAMPLE\_RANDOM，表示对系统熵有贡献，对系统获取随机数有好处。(这几个flag是可以通过或的方式同时使用的）。  
**devname**设置中断名称，通常是设备驱动程序的名称  在cat /proc/interrupts中可以看到此名称。  
**dev\_id**在中断共享时会用到，一般设置为这个设备的设备结构体或者NULL。  
request\_irq()返回0表示成功，返回-INVAL表示中断号无效或处理函数指针为NULL，返回-EBUSY表示中断已经被占用且不能共享。

int request\_irq**(**unsigned int irq**,** irq\_handler\_t handler**,**

unsigned long irqflags**,** const char **\***devname**,** void **\***dev\_id**)**

**{**

struct irqaction **\***action**;**

int retval**;**

//shared interrupts must pass in a real dev-ID,

**if** **((**irqflags **&** IRQF\_SHARED**)** **&&** **!**dev\_id**)**

**return** **-**EINVAL**;**

**if** **(**irq **>=** NR\_IRQS**)**

**return** **-**EINVAL**;**

//该中断号已经被使用，并且未共享

**if** **(**irq\_desc**[**irq**].**status **&** IRQ\_NOREQUEST**)**

**return** **-**EINVAL**;**

**if** **(!**handler**)**

**return** **-**EINVAL**;**

//分配了一个结构，结构中的成员指向传递进来的参数

action **=** kmalloc**(sizeof(**struct irqaction**),** GFP\_ATOMIC**);**

**if** **(!**action**)**

**return** **-**ENOMEM**;**

action**->**handler **=** handler**;**

action**->**flags **=** irqflags**;**

cpus\_clear**(**action**->**mask**);**

action**->**name **=** devname**;**

action**->**next **=** **NULL;**

action**->**dev\_id **=** dev\_id**;**

select\_smp\_affinity**(**irq**);**

//设置irq

//1)irq\_des[irq] 已irq为下标找到数组项

//2)在irq\_des[irq]链表里面加入传递进来的参数action

//3)desc->chip->settype()设置为中断引脚

//4)desc->chip->startup / desc->chip->enable 使能中断

retval **=** setup\_irq**(**irq**,** action**);**

**if** **(**retval**)**

kfree**(**action**);**

**return** retval**;**

**}**

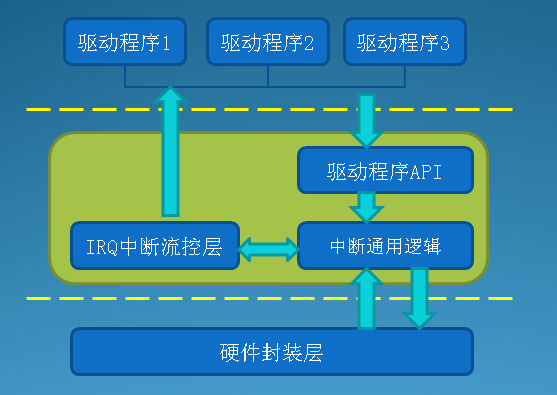
从源码看request\_irq会使能该中断。

### 1.5 中断卸载

卸载中断处理函数通过free\_irq函数来实现。  
free\_irq函数的处理过程与request\_irq函数相反  
(1)根据中断号irq，dev\_id从action链表中找到表项，将它移除。  
(2)如果它是惟一的表项，还要调用IRQ\_DESC[IRQ].CHIP->SHUTDOWN 或IRQ\_DESC[IRQ].CHIP->DISABLW来关闭中断。

### 1.6 外部中断处理流程分析

#### 1.6.1 中断基本架构



**(1)硬件封装层**  
它包含了体系架构相关的所有代码，包括中断控制器的抽象封装，arch相关的中断初始化，以及各个IRQ的相关数据结构的初始化工作，cpu的中断入口也会在arch相关的代码中实现。中断通用逻辑层通过标准的封装接口（实际上就是struct irq\_chip定义的接口）访问并控制中断控制器的行为，体系相关的中断入口函数在获取IRQ编号后，通过中断通用逻辑层提供的标准函数，把中断调用传递到中断流控层中。

**(2)中断流控层**所谓中断流控是指合理并正确地处理连续发生的中断，比如一个中断在处理中，同一个中断再次到达时如何处理，何时应该屏蔽中断，何时打开中断，何时回应中断控制器等一系列的操作。该层实现了与体系和硬件无关的中断流控处理操作，它针对不同的中断电气类型（level，edge......），实现了对应的标准中断流控处理函数，在这些处理函数中，最终会把中断控制权传递到驱动程序注册中断时传入的处理函数或者是中断线程中。目前内核提供了以下几个主要的中断流控函数的实现（只列出部分）：

handle\_simple\_irq**();**

handle\_level\_irq**();** 电平中断流控处理程序

handle\_edge\_irq**();** 边沿触发中断流控处理程序

handle\_fasteoi\_irq**();** 需要eoi的中断处理器使用的中断流控处理程序

handle\_percpu\_irq**();** 该irq只有单个cpu响应时使用的流控处理程序

**(3)中断通用逻辑层**

该层实现了对中断系统几个重要数据的管理，并提供了一系列的辅助管理函数。同时，该层还实现了中断线程的实现和管理，共享中断和嵌套中断的实现和管理，另外它还提供了一些接口函数，它们将作为硬件封装层和中断流控层以及驱动程序API层之间的桥梁，例如以下API：

generic\_handle\_irq**();**

irq\_to\_desc**();**

irq\_set\_chip**();**

irq\_set\_chained\_handler**();**

**(4)驱动程序API**  
该部分向驱动程序提供了一系列的API，用于向系统申请/释放中断，打开/关闭中断，设置中断类型和中断唤醒系统的特性等操作。驱动程序的开发者通常只会使用到这一层提供的这些API即可完成驱动程序的开发工作，其他的细节都由另外几个软件层较好地“隐藏”起来了，驱动程序开发者无需再关注底层的实现，这看起来确实是一件美妙的事情，不过我认为，要想写出好的中断代码，还是花点时间了解一下其他几层的实现吧。其中的一些API如下：

enable\_irq**();**

disable\_irq**();**

disable\_irq\_nosync**();**

request\_threaded\_irq**();**

request\_irq**();**

irq\_set\_affinity**();**

#### 1.6.2 外部中断0

##### 1.6.2.1 外部中断0的初始化

**for** **(**irqno **=** IRQ\_EINT0**;** irqno **<=** IRQ\_EINT3**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (ext int)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_eint0t4**);**

set\_irq\_handler**(**irqno**,** handle\_edge\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

**(1)设置irqno对应的irq\_chip**

irq\_chip结构体定义如下：

struct irq\_chip **{**

const char **\***name**;**

unsigned int **(\***startup**)(**unsigned int irq**);**

void **(\***shutdown**)(**unsigned int irq**);**

void **(\***enable**)(**unsigned int irq**);**

void **(\***disable**)(**unsigned int irq**);**

void **(\***ack**)(**unsigned int irq**);**

void **(\***mask**)(**unsigned int irq**);**

void **(\***mask\_ack**)(**unsigned int irq**);**

void **(\***unmask**)(**unsigned int irq**);**

void **(\***eoi**)(**unsigned int irq**);**

void **(\***end**)(**unsigned int irq**);**

void **(\***set\_affinity**)(**unsigned int irq**,** cpumask\_t dest**);**

int **(\***retrigger**)(**unsigned int irq**);**

int **(\***set\_type**)(**unsigned int irq**,** unsigned int flow\_type**);**

int **(\***set\_wake**)(**unsigned int irq**,** unsigned int on**);**

const char **\***typename**;**

**};**

s3c\_irq\_eint0t4的定义为：

static struct irq\_chip s3c\_irq\_eint0t4 **=** **{**

**.**name **=** "s3c-ext0"**,**

**.**ack **=** s3c\_irq\_ack**,**

**.**mask **=** s3c\_irq\_mask**,**

**.**unmask **=** s3c\_irq\_unmask**,**

**.**set\_wake **=** s3c\_irq\_wake**,**

**.**set\_type **=** s3c\_irqext\_type**,**

**};**

s3c\_irq\_ack用于响应中断，通常是清除当前中断使得可以接收下一个中断。  
通过写SRCPND，INTPND对应的bit位，清空当前中断。

static inline void

s3c\_irq\_ack**(**unsigned int irqno**)**

**{**

unsigned long bitval **=** 1UL **<<** **(**irqno **-** IRQ\_EINT0**);**

\_\_raw\_writel**(**bitval**,** S3C2410\_SRCPND**);**

\_\_raw\_writel**(**bitval**,** S3C2410\_INTPND**);**

**}**

s3c\_irq\_mask通过操作INTMSK用于屏蔽对应的中断。

static void

s3c\_irq\_mask**(**unsigned int irqno**)**

**{**

unsigned long mask**;**

irqno **-=** IRQ\_EINT0**;**

mask **=** \_\_raw\_readl**(**S3C2410\_INTMSK**);**

mask **|=** 1UL **<<** irqno**;**

\_\_raw\_writel**(**mask**,** S3C2410\_INTMSK**);**

**}**

s3c\_irq\_unmask操作INTMSK打开中断使能。

static void

s3c\_irq\_unmask**(**unsigned int irqno**)**

**{**

unsigned long mask**;**

**if** **(**irqno **!=** IRQ\_TIMER4 **&&** irqno **!=** IRQ\_EINT8t23**)**

irqdbf2**(**"s3c\_irq\_unmask %d\n"**,** irqno**);**

irqno **-=** IRQ\_EINT0**;**

mask **=** \_\_raw\_readl**(**S3C2410\_INTMSK**);**

mask **&=** **~(**1UL **<<** irqno**);**

\_\_raw\_writel**(**mask**,** S3C2410\_INTMSK**);**

**}**

s3c\_irq\_wake用于设置硬件唤醒终端源。内核允许EINT0--3外部中断将其唤醒。

int

s3c\_irq\_wake**(**unsigned int irqno**,** unsigned int state**)**

**{**

unsigned long irqbit **=** 1 **<<** **(**irqno **-** IRQ\_EINT0**);**

**if** **(!(**s3c\_irqwake\_intallow **&** irqbit**))**

**return** **-**ENOENT**;**

printk**(**KERN\_INFO "wake %s for irq %d\n"**,**

state **?** "enabled" **:** "disabled"**,** irqno**);**

**if** **(!**state**)**

s3c\_irqwake\_intmask **|=** irqbit**;**

**else**

s3c\_irqwake\_intmask **&=** **~**irqbit**;**

**return** 0**;**

**}**

s3c\_irqext\_type用来设置对应外部中断寄存器，设置gpio为外部中断模式，设置外部中断触发模式。

int

s3c\_irqext\_type**(**unsigned int irq**,** unsigned int type**)**

**{**

void \_\_iomem **\***extint\_reg**;**

void \_\_iomem **\***gpcon\_reg**;**

unsigned long gpcon\_offset**,** extint\_offset**;**

unsigned long newvalue **=** 0**,** value**;**

**if** **((**irq **>=** IRQ\_EINT0**)** **&&** **(**irq **<=** IRQ\_EINT3**))**

**{**

gpcon\_reg **=** S3C2410\_GPFCON**;**

extint\_reg **=** S3C24XX\_EXTINT0**;**

gpcon\_offset **=** **(**irq **-** IRQ\_EINT0**)** **\*** 2**;**

extint\_offset **=** **(**irq **-** IRQ\_EINT0**)** **\*** 4**;**

**}**

**else** **if** **((**irq **>=** IRQ\_EINT4**)** **&&** **(**irq **<=** IRQ\_EINT7**))**

**{**

gpcon\_reg **=** S3C2410\_GPFCON**;**

extint\_reg **=** S3C24XX\_EXTINT0**;**

gpcon\_offset **=** **(**irq **-** **(**EXTINT\_OFF**))** **\*** 2**;**

extint\_offset **=** **(**irq **-** **(**EXTINT\_OFF**))** **\*** 4**;**

**}**

**else** **if** **((**irq **>=** IRQ\_EINT8**)** **&&** **(**irq **<=** IRQ\_EINT15**))**

**{**

gpcon\_reg **=** S3C2410\_GPGCON**;**

extint\_reg **=** S3C24XX\_EXTINT1**;**

gpcon\_offset **=** **(**irq **-** IRQ\_EINT8**)** **\*** 2**;**

extint\_offset **=** **(**irq **-** IRQ\_EINT8**)** **\*** 4**;**

**}**

**else** **if** **((**irq **>=** IRQ\_EINT16**)** **&&** **(**irq **<=** IRQ\_EINT23**))**

**{**

gpcon\_reg **=** S3C2410\_GPGCON**;**

extint\_reg **=** S3C24XX\_EXTINT2**;**

gpcon\_offset **=** **(**irq **-** IRQ\_EINT8**)** **\*** 2**;**

extint\_offset **=** **(**irq **-** IRQ\_EINT16**)** **\*** 4**;**

**}** **else**

**return** **-**1**;**

//设置gpio为外部中断模式

value **=** \_\_raw\_readl**(**gpcon\_reg**);**

value **=** **(**value **&** **~(**3 **<<** gpcon\_offset**))** **|** **(**0x02 **<<** gpcon\_offset**);**

\_\_raw\_writel**(**value**,** gpcon\_reg**);**

/\*000 = Low level 001 = High level 01x = Falling edge triggered

10x = Rising edge triggered 11x = Both edge triggered\*/

**switch** **(**type**)**

**{**

**case** IRQT\_NOEDGE**:**

printk**(**KERN\_WARNING "No edge setting!\n"**);**

**break;**

**case** IRQT\_RISING**:**

newvalue **=** S3C2410\_EXTINT\_RISEEDGE**;**

**break;**

**case** IRQT\_FALLING**:**

newvalue **=** S3C2410\_EXTINT\_FALLEDGE**;**

**break;**

**case** IRQT\_BOTHEDGE**:**

newvalue **=** S3C2410\_EXTINT\_BOTHEDGE**;**

**break;**

**case** IRQT\_LOW**:**

newvalue **=** S3C2410\_EXTINT\_LOWLEV**;**

**break;**

**case** IRQT\_HIGH**:**

newvalue **=** S3C2410\_EXTINT\_HILEV**;**

**break;**

**default:**

printk**(**KERN\_ERR "No such irq type %d"**,** type**);**

**return** **-**1**;**

**}**

value **=** \_\_raw\_readl**(**extint\_reg**);**

value **=** **(**value **&** **~(**7 **<<** extint\_offset**))** **|** **(**newvalue **<<** extint\_offset**);**

\_\_raw\_writel**(**value**,** extint\_reg**);**

**return** 0**;**

**}**

同时除了这些设置，还有一些默认的选项会一起被初始化。

void irq\_chip\_set\_defaults**(**struct irq\_chip **\***chip**)**

**{**

**if** **(!**chip**->**enable**)**

chip**->**enable **=** default\_enable**;**

**if** **(!**chip**->**disable**)**

chip**->**disable **=** default\_disable**;**

**if** **(!**chip**->**startup**)**

chip**->**startup **=** default\_startup**;**

**if** **(!**chip**->**shutdown**)**

chip**->**shutdown **=** chip**->**disable**;**

**if** **(!**chip**->**name**)**

chip**->**name **=** chip**->**typename**;**

**if** **(!**chip**->**end**)**

chip**->**end **=** dummy\_irq\_chip**.**end**;**

**}**

其中default\_enable调用umask打开中断。

static void default\_enable**(**unsigned int irq**)**

**{**

struct irq\_desc **\***desc **=** irq\_desc **+** irq**;**

desc**->**chip**->**unmask**(**irq**);**

desc**->**status **&=** **~**IRQ\_MASKED**;**

**}**

default\_startup 和default\_enable一致。

static unsigned int default\_startup**(**unsigned int irq**)**

**{**

irq\_desc**[**irq**].**chip**->**enable**(**irq**);**

**return** 0**;**

**}**

由此可见，irq\_chip结构体设置了许多和平台相关的硬件寄存器操作函数。

**(2) set\_irq\_handler(irqno, handle\_edge\_irq)**

设置了中断流控函数为handle\_edge\_irq，后面会分析该函数。

**(3)set\_irq\_flags(irqno, IRQF\_VALID)**

设置该中断可以被request。

void set\_irq\_flags**(**unsigned int irq**,** unsigned int iflags**)**

**{**

struct irq\_desc **\***desc**;**

unsigned long flags**;**

**if** **(**irq **>=** NR\_IRQS**)** **{**

printk**(**KERN\_ERR "Trying to set irq flags for IRQ%d\n"**,** irq**);**

**return;**

**}**

desc **=** irq\_desc **+** irq**;**

spin\_lock\_irqsave**(&**desc**->**lock**,** flags**);**

//首先设置中断状态为不能被request,不能被probe，在request的时候不能被自动enable

desc**->**status **|=** IRQ\_NOREQUEST **|** IRQ\_NOPROBE **|** IRQ\_NOAUTOEN**;**

//如果设置了IRQF\_VALID，说明该中断可以被request

**if** **(**iflags **&** IRQF\_VALID**)**

desc**->**status **&=** **~**IRQ\_NOREQUEST**;**

//如果设置了IRQF\_PROBE,说明了该中断可以probe

**if** **(**iflags **&** IRQF\_PROBE**)**

desc**->**status **&=** **~**IRQ\_NOPROBE**;**

//如果没有设置IRQF\_NOAUTOEN，则request中断时，该中断自动使能

**if** **(!(**iflags **&** IRQF\_NOAUTOEN**))**

desc**->**status **&=** **~**IRQ\_NOAUTOEN**;**

spin\_unlock\_irqrestore**(&**desc**->**lock**,** flags**);**

**}**

##### 1.6.2.2 外部中断0的注册

request\_irq**(**IRQ\_EINT0**,** buttons\_irq**,** IRQT\_BOTHEDGE**,** "S2"**,** **&**pins\_desc**[**0**]);**

button\_irq为驱动注册的中断处理函数，由于可能是多个外部中断共享一个，根据dev\_id区分是哪个中断发生了。

static irqreturn\_t buttons\_irq**(**int irq**,** void **\***dev\_id**)**

**{**

struct pin\_desc **\*** pindesc **=** **(**struct pin\_desc **\*)**dev\_id**;**

unsigned int pinval**;**

pinval **=** s3c2410\_gpio\_getpin**(**pindesc**->**pin**);**

**if** **(**pinval**)**

**{**

/\* 松开 \*/

key\_val **=** 0x80 **|** pindesc**->**key\_val**;**

**}**

**else**

**{**

/\* 按下 \*/

key\_val **=** pindesc**->**key\_val**;**

**}**

ev\_press **=** 1**;** /\* 表示中断发生了 \*/

wake\_up\_interruptible**(&**button\_waitq**);** /\* 唤醒休眠的进程 \*/

**return** IRQ\_RETVAL**(**IRQ\_HANDLED**);**

**}**

request\_irq主要做了如下操作：  
(1)irq\_des[irq] 已irq为下标找到数组项。  
(2)在irq\_des[irq]链表里面加入传递进来的参数action。  
(3)desc->chip->settype()设置为中断引脚。  
(4)desc->chip->startup / desc->chip->enable 使能中断。

**那么对IRQ\_EINT0执行request\_irq动作，完成的操作如下：**(1) irq\_des[IRQ\_EINT0]链表加入buttons\_irq节点。  
(2)如果设置了IRQT\_BOTHEDGE，则会调用s3c\_irqext\_type设置中断方式。  
(3)如果没有设置IRQ\_NOAUTOEN，则会调用default\_startup打开中断。

##### 1.6.2.3 外部中断0的响应

当外部中断0发生时，整个中断处理流程如下：  
(1)cpu进入IRQ异常处理模式。  
(2)中断向量表中通过b vector\_irq + stubs\_offset跳转到vector\_irq  
(3)保存相关现场后进入\_irq\_user，并切换到svc模式  
(4) asm\_do\_IRQ  
(5) irq\_des[irq]以中断号为下标取出一项 ->handle\_irq。  
(6) handle\_irq = handle\_edge\_irq进入handle\_edge\_irq。

**下面将详细分析handle\_edge\_irq函数**  
handle\_edge\_irq处理边沿触发中断的流控操作，只有设备的中断请求引脚（中断线）的电平发生跳变时（由高变低或者有低变高），才会发出中断请求。因为跳变是一瞬间，而且不会像电平中断能保持住电平，所以处理不当就特别容易漏掉一次中断请求，为了避免这种情况，屏蔽中断的时间必须越短越好。内核的开发者们显然意识到这一点，在正是处理中断前，判断IRQ\_PROGRESS标志没有被设置的情况下，只是ack irq，并没有mask irq，以便复位设备的中断请求引脚，在这之后的中断处理期间，另外的cpu可以再次响应同一个irq请求，如果IRQ\_PROGRESS已经置位，表明另一个CPU正在处理该irq的上一次请求，这种情况下，他只是简单地设置IRQ\_PENDING标志，然后mask\_ack\_irq后退出，中断请求交由原来的CPU继续处理。因为是mask\_ack\_irq，所以系统实际上只允许挂起一次中断。

从上面的分析可以知道，处理中断期间，另一次请求可能由另一个cpu响应后挂起，所以在处理完本次请求后还要判断IRQS\_PENDING标志，如果被置位，当前cpu要接着处理被另一个cpu“委托”的请求。内核在这里设置了一个循环来处理这种情况，直到IRQ\_PENDING标志无效为止，而且因为另一个cpu在响应并挂起irq时，会mask irq，所以在循环中要再次unmask irq，以便另一个cpu可以再次响应并挂起irq。

void fastcall

handle\_edge\_irq**(**unsigned int irq**,** struct irq\_desc **\***desc**)**

**{**

const unsigned int cpu **=** smp\_processor\_id**();**

spin\_lock**(&**desc**->**lock**);**

//清空IRQ\_REPLAY和IRQ\_WAITING标志

desc**->**status **&=** **~(**IRQ\_REPLAY **|** IRQ\_WAITING**);**

//假如这个irq已经在处理或者没有被使能，或者没有设置处理函数

//则屏蔽掉并清空这个中断并，最后退出。

**if** **(**unlikely**((**desc**->**status **&** **(**IRQ\_INPROGRESS **|** IRQ\_DISABLED**))** **||**

**!**desc**->**action**))** **{**

desc**->**status **|=** **(**IRQ\_PENDING **|** IRQ\_MASKED**);**

mask\_ack\_irq**(**desc**,** irq**);**

**goto** out\_unlock**;**

**}**

//发生中断的次数

kstat\_cpu**(**cpu**).**irqs**[**irq**]++;**

//清空当前中断

desc**->**chip**->**ack**(**irq**);**

//标志该irq正在处理中

desc**->**status **|=** IRQ\_INPROGRESS**;**

**do** **{**

struct irqaction **\***action **=** desc**->**action**;**

irqreturn\_t action\_ret**;**

//判断链表是否为空

**if** **(**unlikely**(!**action**))** **{**

desc**->**chip**->**mask**(**irq**);**

**goto** out\_unlock**;**

**}**

//如果在处理该irq请求的同时，irq再次到来，上面会屏蔽掉该中断

//所以如果发现IRQ\_PENDING标志，则需要重新打开中断。

**if** **(**unlikely**((**desc**->**status **&**

**(**IRQ\_PENDING **|** IRQ\_MASKED **|** IRQ\_DISABLED**))** **==**

**(**IRQ\_PENDING **|** IRQ\_MASKED**)))** **{**

desc**->**chip**->**unmask**(**irq**);**

desc**->**status **&=** **~**IRQ\_MASKED**;**

**}**

desc**->**status **&=** **~**IRQ\_PENDING**;**

spin\_unlock**(&**desc**->**lock**);**

//真正的处理过程

action\_ret **=** handle\_IRQ\_event**(**irq**,** action**);**

**if** **(!**noirqdebug**)**

note\_interrupt**(**irq**,** desc**,** action\_ret**);**

spin\_lock**(&**desc**->**lock**);**

//发生期间有中断，则再次执行。

**}** **while** **((**desc**->**status **&** **(**IRQ\_PENDING **|** IRQ\_DISABLED**))** **==** IRQ\_PENDING**);**

desc**->**status **&=** **~**IRQ\_INPROGRESS**;**

out\_unlock**:**

spin\_unlock**(&**desc**->**lock**);**

**}**

(7) handle\_IRQ\_event逐个调用aciton链表中注册的中断处理函数，也就是会调用buttons\_irq函数。

#### 1.6.3 外部中断11

外部中断11有两级中断，所以在处理流程上和外部中断0还是有一些不太一样。

##### 1.6.3.1 外部中断11的初始化

因为外部中断11并不是一个单独的中断号，所以要先设置它的父中断EINT8t23。因为当中断11发生时，实际上得到的中断是EINT8t23。

**(1)EINT8t23的相关设置如下：**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_level\_chip**);**

set\_irq\_chained\_handler**(**IRQ\_EINT8t23**,** s3c\_irq\_demux\_extint8**);**

s3c\_irq\_level\_chip和中断0的和s3c\_irq\_eint0t4结构体在ack选项上的设置不一样，并且没有set\_type选项。

struct irq\_chip s3c\_irq\_level\_chip **=** **{**

**.**name **=** "s3c-level"**,**

**.**ack **=** s3c\_irq\_maskack**,**

**.**mask **=** s3c\_irq\_mask**,**

**.**unmask **=** s3c\_irq\_unmask**,**

**.**set\_wake **=** s3c\_irq\_wake

**};**

中断发生时候响应中断的函数为s3c\_irq\_maskack。  
除了清空相关的中断，还会是暂时屏蔽掉该中断。

static inline void

s3c\_irq\_maskack**(**unsigned int irqno**)**

**{**

unsigned long bitval **=** 1UL **<<** **(**irqno **-** IRQ\_EINT0**);**

unsigned long mask**;**

mask **=** \_\_raw\_readl**(**S3C2410\_INTMSK**);**

\_\_raw\_writel**(**mask**|**bitval**,** S3C2410\_INTMSK**);**

\_\_raw\_writel**(**bitval**,** S3C2410\_SRCPND**);**

\_\_raw\_writel**(**bitval**,** S3C2410\_INTPND**);**

**}**

接下来会设置EINT8t23中断处理函数，注意这里使用的set\_irq\_chained\_handler而不是set\_irq\_handler，set\_irq\_chained\_handler在设置中断处理函数的同时，还会打开该中断，因为在驱动层并不会request该irq。

**(2)EINT13的相关初始化如下：**

**for** **(**irqno **=** IRQ\_EINT4**;** irqno **<=** IRQ\_EINT23**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (extended s3c irq)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irqext\_chip**);**

set\_irq\_handler**(**irqno**,** handle\_edge\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

EINT13由于需要操作子中断寄存器，所以irq\_chip结构体也会不太一样。

static struct irq\_chip s3c\_irqext\_chip **=** **{**

**.**name **=** "s3c-ext"**,**

**.**mask **=** s3c\_irqext\_mask**,**

**.**unmask **=** s3c\_irqext\_unmask**,**

**.**ack **=** s3c\_irqext\_ack**,**

**.**set\_type **=** s3c\_irqext\_type**,**

**.**set\_wake **=** s3c\_irqext\_wake

**};**

这些函数由于是外部子中断，需要操作EINTMASK，EINTPEND等寄存器，具体代码不做分析，和EINT0的操作类似。

后面两句代码设置了EINT13的操作函数handle\_edge\_irq，并设置该irq能被request。

##### 1.6.3.2 外部中断11的注册

request\_irq**(**IRQ\_EINT11**,** buttons\_irq**,** IRQT\_BOTHEDGE**,** "S4"**,** **&**pins\_desc**[**2**]);**

为EINT11注册了驱动操作函数buttons\_irq，和EINT0注册是一样的。设置好中断模式并打开中断。

##### 1.6.3.3 外部中断11的响应

当外部中断11发生时，整个中断处理流程如下：  
(1)cpu中断发生，中断号为EINT8t23  
(2)中断向量表中通过b vector\_irq + stubs\_offset跳转到vector\_irq  
(3)保存相关现场后进入\_irq\_user，并切换到svc模式  
(4) asm\_do\_IRQ  
(5) irq\_des[irq]以中断号为下标取出一项 ->handle\_irq。  
(6) handle\_irq = s3c\_irq\_demux\_extint8进入s3c\_irq\_demux\_extint8。

下面将分析s3c\_irq\_demux\_extint8函数：

static void

s3c\_irq\_demux\_extint8**(**unsigned int irq**,**

struct irq\_desc **\***desc**)**

**{**

//EINT8-EINT23 发生时，相应位被置1

unsigned long eintpnd **=** \_\_raw\_readl**(**S3C24XX\_EINTPEND**);**

//EINT8-EINT23 屏蔽寄存器

unsigned long eintmsk **=** \_\_raw\_readl**(**S3C24XX\_EINTMASK**);**

//清除被屏蔽的位

eintpnd **&=** **~**eintmsk**;**

//清除低8位，这里处理的是外部中断8到23

eintpnd **&=** **~**0xff**;**

**while** **(**eintpnd**)** **{**

//确定eintpnd中为1的最高位

irq **=** \_\_ffs**(**eintpnd**);**

//将此位清0

eintpnd **&=** **~(**1**<<**irq**);**

//重新计算中断号，因为中断4开始，就是有串联中断了

irq **+=** **(**IRQ\_EINT4 **-** 4**);**

//调用真正的处理函数

desc\_handle\_irq**(**irq**,** irq\_desc **+** irq**);**

**}**

**}**

(7)调用EINT11的handle\_irq=handle\_edge\_irq。

(8)后面的过程和EINT0的处理一样。

### 1.6.4 handle\_level\_irq

电平中断的特点是，只要设备的中断请求引脚（中断线）保持在预设的触发电平，中断就会一直被请求，所以，为了避免同一中断被重复响应，必须在处理中断前先把mask irq，然后ack irq，以便复位设备的中断请求引脚，响应完成后再unmask irq。

### 1.7 顶半部和底半部

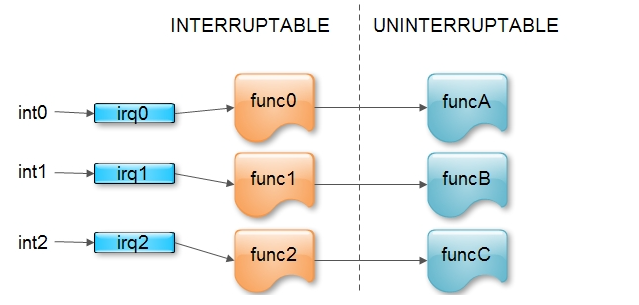
如果我们将Linux下中断处理分成传说中的顶半部和底半部，那么所谓的“顶半部”，大约就是指实际响应中断的例程，也就是用request\_irq注册的中断处理函数，而所谓的“底半部”是一个被“顶半部”调用，并在稍后更安全的时间内执行的例程。

中断处理例程是在中断时间内运行的，它的行为会受到一些限制。处理例程不能向用户空间发送或者接收数据，因为它不是任何进程的上下文中执行的，处理例程不能做任何可能发生休眠的操作，例如调用wait\_event,使用不带GFP\_ATOMIC标志的内存分配操作，或者锁住一个信号量等等，不能调用schdule函数。

中断处理例程的一个典型任务是：如果中断通知进程所等待的事件已发生，就会唤醒在该设备上休眠的进程。

Linux内核中断机制：为了在中断执行时间尽可能短和中断处理需要完成大量工作之间找到一个平衡点，Linux将中断处理程序分解为两个半部，顶半部和底半部。

随着系统的不断复杂，中断处理函数要做的事情也越来越多，多到都来不及接收新的中断了。于是发生了中断丢失，这显然不行，于是产生了新的机制：分离中断接收与中断处理过程。中断接收在屏蔽中断的情况下完成；中断处理在时能中断的情况下完成，这部分被称为中断下半部。

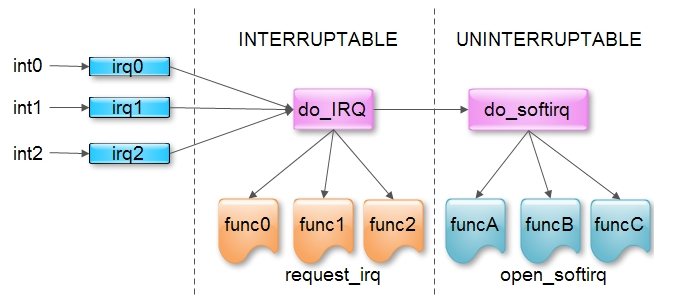


从上图中看，只看int0的处理。Func0为中断接收函数。中断只能简单的触发func0，而func0则能做更多的事情，它与funcA之间可以使用队列等缓存机制。当又有中断发生时，func0被触发，然后发送一个中断请求到缓存队列，然后让funcA去处理。由于func0做的事情是很简单的，所以不会影响int0的再次接收。而且在func0返回时就会使能int0，因此funcA执行时间再长也不会影响int0的接收。

作为一个操作系统显然不能任由每个中断都各自为政，统一管理是必须的。下面是三种底半部的实现方式：

#### 1.7.1 软中断

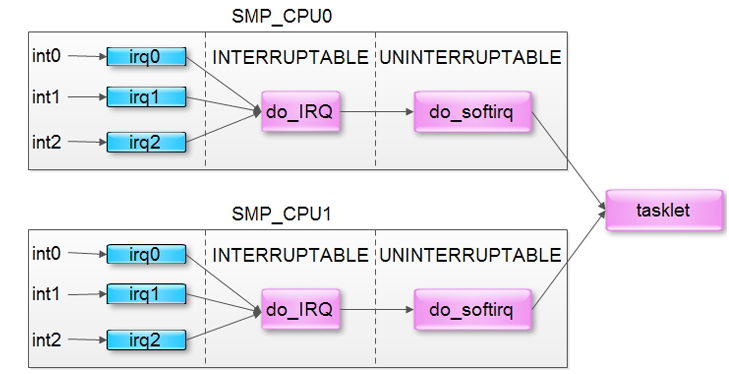
中断共同部分放在函数do\_IRQ中，需要添加中断处理函数时，通过request\_irq实现。下半部放在do\_softirq中，也就是软中断，通过open\_softirq添加对应的处理函数。



#### 1.7.2 tasklet

旧事物跟不上历史的发展时，总会有新事物出现。随着中断数的不停增加，软中断不够用了，于是下半部又做了进化。软中断用轮询的方式处理。假如正好是最后一种中断，则必须循环完所有的中断类型，才能最终执行对应的处理函数。显然当年开发人员为了保证轮询的效率，于是限制中断个数为32个。

为了提高中断处理数量，顺道改进处理效率，于是产生了tasklet机制。  
Tasklet采用无差别的队列机制，有中断时才执行，免去了循环查表之苦。  
CDMA因为频谱重叠问题，必须降功率。而功率降低后，又发现原来功率低了有助于环保。  
Tasklet作为一种新机制，显然可以承担更多的优点。正好这时候SMP越来越火了，因此又在tasklet中加入了SMP机制，保证同种中断只在一个cpu上执行。在软中断时代，显然没有这种考虑。因此同一种中断可以在两个cpu上同时执行，很可能造成冲突。



总结下tasklet的优点：  
（1）无类型数量限制；  
（2）效率高，无需循环查表；  
（3）支持SMP机制；

**tasklet的使用方法：**

//定义与绑定tasklet函数

 void test\_tasklet\_action**(**unsigned long t**);**

 DECLARE\_TASKLET**(**test\_tasklet**,** test\_tasklet\_action**,** 0**);**

 //中断处理底半部

 void test\_tasklet\_action**(**unsigned long t**)**

**{**

     printk**(**"tasklet is executing\n"**);**

**}**

/\*中断处理顶半部\*/

static irqreturn\_t xxx\_interrupt**(**int irq**,** void **\***dev\_id**)**

**{**

**.....**

    tasklet\_schedule**(&**test\_tasklet**);**

**.....**

**}**

/\*设备驱动加载模块\*/

int \_\_init xxx\_init**(**void**)**

**{**

**......**

    request\_irq**(**IRQ\_EINT0**,**xxx\_interrupt**,** IRQ\_TYPE\_LEVEL\_LOW**,** "xxx"**,** **NULL);**

**......**

**}**

/\*设备驱动卸载模块\*/

void \_\_exit xxx\_exit**(**void**)**

**{**

**......**

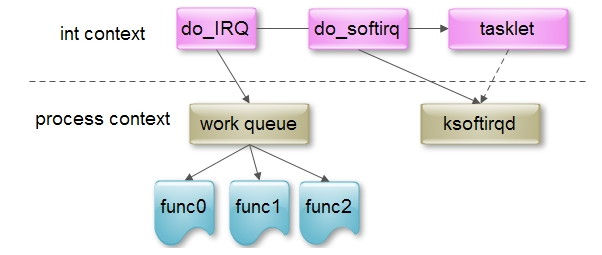
    free**(**IRQ\_EINT0**,NULL);**

**......**

**}**

#### 1.7.3 工作队列

前面的机制不论如何折腾，有一点是不会变的。它们都在中断上下文中。什么意思？说明它们不可挂起。而且由于是串行执行，因此只要有一个处理时间较长，则会导致其他中断响应的延迟。为了完成这些不可能完成的任务，于是出现了工作队列。工作队列说白了就是一组内核线程，作为中断守护线程来使用。多个中断可以放在一个线程中，也可以每个中断分配一个线程。工作队列对线程作了封装，使用起来更方便。  
因为工作队列是线程，所以我们可以使用所有可以在线程中使用的方法。



**工作队列的使用方法：**

(1)利用系统共享的工作队列添加工作：

struct work\_struct my\_wq**;**/\*定义一个工作队列\*/

void my\_wq\_func**(**unsigned long**);**/\*定义一个处理函数\*/

//中断处理底半部

void my\_wq\_func**(**unsigned long t**)**

**{**

**.......**

**}**

/\*中断处理顶半部\*/

static irqreturn\_t xxx\_interrupt**(**int irq**,** void **\***dev\_id**)**

**{**

**.....**

    schedule\_work**(&** my\_wq**);**

**.....**

**}**

/\*设备驱动加载模块\*/

int \_\_init xxx\_init**(**void**)**

**{**

**......**

    request\_irq**(**IRQ\_EINT0**,**xxx\_interrupt**,** IRQ\_TYPE\_LEVEL\_LOW**,** "xxx"**,** **NULL);**

    INIT\_WORK**(&**my\_wq**,(**void **(\*)(**void**\*))**my\_wq\_func**,NULL);**/\*初始化工作队列并将其与处理函数绑定\*/

**......**

**}**

/\*设备驱动卸载模块\*/

void \_\_exit xxx\_exit**(**void**)**

**{**

**......**

    free**(**IRQ\_EINT0**,NULL);**

**......**

**}**

(2) 创建自己的工作队列来添加工作

//第一步：声明工作处理函数和一个指向工作队列的指针

void my\_func**();**

struct workqueue\_struct **\***p\_queue**;**

//第二步：创建自己的工作队列和工作结构体变量(通常在open函数中完成)

p\_queue**=**create\_workqueue**(**"my\_queue"**);**

struct work\_struct my\_work**;**

INIT\_WORK**(&**my\_work**,** my\_func**,** **&**data**);**

//第三步：将工作添加入自己创建的工作队列等待执行

queue\_work**(**p\_queue**,** **&**my\_work**);**

//第四步：删除自己的工作队列,一般是在close函数中删除

destroy\_workqueue**(**p\_queue**);**

### 1.8 中断子系统其他知识

#### 1.8.1 irq的打开和关闭

中断子系统为我们提供了一系列用于irq的打开和关闭的函数接口，其中最基本的一对是：

disable\_irq**(**unsigned int irq**);**

enable\_irq**(**unsigned int irq**);**

这两个API应该配对使用，disable\_irq可以被多次嵌套调用，要想重新打开irq，enable\_irq必须也要被调用同样的次数，为此，irq\_desc结构中的depth字段专门用于这两个API嵌套深度的管理。当某个irq首次被驱动程序申请时，默认情况下，设置depth的初始值是0，对应的irq处于打开状态。

void enable\_irq**(**unsigned int irq**)**

**{**

struct irq\_desc **\***desc **=** irq\_desc **+** irq**;**

unsigned long flags**;**

**if** **(**irq **>=** NR\_IRQS**)**

**return;**

spin\_lock\_irqsave**(&**desc**->**lock**,** flags**);**

**switch** **(**desc**->**depth**)** **{**

**case** 0**:**

printk**(**KERN\_WARNING "Unbalanced enable for IRQ %d\n"**,** irq**);**

WARN\_ON**(**1**);**

**break;**

**case** 1**:** **{**

unsigned int status **=** desc**->**status **&** **~**IRQ\_DISABLED**;**

/\* Prevent probing on this irq: \*/

desc**->**status **=** status **|** IRQ\_NOPROBE**;**

check\_irq\_resend**(**desc**,** irq**);**

/\* fall-through \*/

**}**

**default:**

desc**->**depth**--;**

**}**

spin\_unlock\_irqrestore**(&**desc**->**lock**,** flags**);**

**}**

当depth的值为1时，才真正地调用irq\_enable()，它最终通过chip->unmask或chip->enable回调开启中断控制器中相应的中断线，如果depth不是1，只是简单地减去1。如果已经是0，驱动还要调用enable\_irq，说明驱动程序处理不当，造成enable与disable不平衡，内核会打印一句警告信息：Unbalanced enable for IRQ xxx。

void disable\_irq**(**unsigned int irq**)**

**{**

struct irq\_desc **\***desc **=** irq\_desc **+** irq**;**

**if** **(**irq **>=** NR\_IRQS**)**

**return;**

disable\_irq\_nosync**(**irq**);**

**if** **(**desc**->**action**)**

synchronize\_irq**(**irq**);**

**}**

void disable\_irq\_nosync**(**unsigned int irq**)**

**{**

struct irq\_desc **\***desc **=** irq\_desc **+** irq**;**

unsigned long flags**;**

**if** **(**irq **>=** NR\_IRQS**)**

**return;**

spin\_lock\_irqsave**(&**desc**->**lock**,** flags**);**

**if** **(!**desc**->**depth**++)** **{**

desc**->**status **|=** IRQ\_DISABLED**;**

desc**->**chip**->**disable**(**irq**);**

**}**

spin\_unlock\_irqrestore**(&**desc**->**lock**,** flags**);**

**}**

#### 1.8.2 共享中断和快速中断

**共享中断**：意思是多个中断共享一个中断号，那么这就意味着，多个设备中断发生时，都会产生一个同样的中断信号。那么怎么在linux下进行检测和处理呢？同时需要注意哪些事项？   
(1)注册共享中断时，request\_irq()函数中的flags标志必须指定为IRQF\_SHARED；   
(2)request\_irq()参数中dev\_id必须唯一；   
(3)共享中断的处理程序中，不能使用disable\_irq(unsigned int irq);   
(4)共享中断的处理程序中，必须对相应中断标志进行检测；

**快速中断**：request\_irq()函数中的flags标志指定为SA\_INTERRUPT，在该中断处理时，其他中断禁止。

## 2. 驱动必备的一些知识

### 2.1 驱动中定时器用法

(1)分配定时器

struct timer\_list mytimer**;**

(2)初始化定时器

init\_timer**(&**mytimer**);** //这个函数内核仅仅初始化它关心的字段。

mytimer**.**expires **=** jiffies **+** 5**\***HZ**;** //超时时间间隔5秒

mytimer**.**function **=** mytimer\_function**;** //指定超时处理函数

mytimer**.**data **=** **(**unsigned long**)&**mydata**;**//给超时处理函数传递的参数

或者

setup\_timer**(&**mytimer**,** mytimer\_function**,** **(**unsigned long**)&**mydata**);**

setup\_timer的定义为：

static inline void setup\_timer**(**struct timer\_list **\*** timer**,**

void **(\***function**)(**unsigned long**),**

unsigned long data**)**

**{**

timer**->**function **=** function**;**

timer**->**data **=** data**;**

init\_timer**(**timer**);**

**}**

(3)将定时器添加到内核中

add\_timer**(&**mytimer**);**

一旦完成添加，内核就开始管理这个定时器，开始计时！一旦超时，内核执行超时处理函数，内核同时将定时器删除掉。注意超时处理函数只执行一次。

超时处理函数最后重新添加定时器重复执行。如果添加如下代码：

mytimer**.**expires **=** jiffies **+** 5**\***HZ**;**

add\_timer**(&**mytimer**);**

如果添加以上代码，由于linux内核支持任务的抢占，中断和SMP，会有并发和竞态问题，导致以上代码执行的结果具有不确定性。以上两个操作步骤不是原子操作！

(4)修改定时器

mod\_timer**(&**mytimer**,** jiffies **+** 3**\***HZ**);**

mod\_timer等价于一下三个步骤：

del\_timer**(&**mytimer**);**

mytimer**.**expires **=** jiffies **+** 3**\***HZ

add\_timer**(&**mytimer**);**

(5)删除定时器

del\_timer**(&**mytimer**);**

如果定时器到期，内核帮你删除，定时器没有到期，可以调用此函数提前将定时器删除。

### 2.2 驱动中延时用法

udelay**(**int n**);**  //延时n微秒

mdelay**(**int n**);**  //延时n毫秒

ndelay**(**int n**);**  //延时n纳秒

实现的原理本质上都是忙等待，ndelay和mdelay都是通过udelay衍生出来的。  
需要包含的头文件：在include/asm-arm/delay.h中定义了udelay（），而在include/[Linux](http://lib.csdn.net/base/linux)/delay.h中定义了mdelay和ndelay。  
对于长时间的忙等待意味这无谓的耗费着cpu的资源，因此对于毫秒级的延时，内核提供了msleep，ssleep等函数，这些函数将使得调用它的进程睡眠参数指定的时间。

void ndelay**(**unsigned long nsecs**);**         //纳秒级：1/10^-10

void udelay**(**unsigned long usecs**);**         //微秒级: 1/10^-6

void mdelay**(**unsigned long msecs**);**         //毫秒级：1/10^-3

### 2.3 驱动中的阻塞之等待队列

在驱动中的read操作中，如果没有数据，需要进入睡眠，数据到来，要唤醒。  
什么是睡眠，其实就是将当前进程的状态设置为 TASK\_INTERRUPTIBLE状态，然后schedule() 让出CPU，让调度器重新选择一个进程来执行。  
等待队列可以用来完成睡眠和唤醒的工作，一个等待队列就是它听起来的样子:一个进程列表, 都等待一个特定的事件。

在 Linux 中, 一个等待队列由一个"等待队列头"来管理, 一个 wait\_queue\_head\_t 类型的结构, 定义在<linux/wait.h>中. 一个等待队列头可被定义和初始化,

DECLARE\_WAIT\_QUEUE\_HEAD**(**name**);**

或者

wait\_queue\_head\_t my\_queue**;**

init\_waitqueue\_head**(&**my\_queue**);**

#### 2.3.1 简单睡眠

Linux 内核中睡眠的最简单方式是一个宏定义, 称为wait\_event(有几个变体); 它结合了处理睡眠的细节和进程在等待的条件的检查. wait\_event 的形式是：

wait\_event**(**queue**,** condition**)**

wait\_event\_interruptible**(**queue**,** condition**)**

wait\_event\_timeout**(**queue**,** condition**,** timeout**)**

wait\_event\_interruptible\_timeout**(**queue**,** condition**,** timeout**)**

如何使用：queue是等待队列头，condition是条件，如果调用wait\_event 前 condition == 0 ，则调用 wait\_event 之后，当前进程就会休眠。  
这四个函数有什么不一样呢？

wait\_event：

将当前进程的状态设置为 TASK\_UNINTERRUPTIBLE ，然后 schedule**()**

wait\_event\_interruptible：

将当前进程的状态设置为TASK\_INTERRUPTIBLE ，然后 schedule**()**

wait\_event\_timeout：

将当前进程的状态设置为TASK\_UNINTERRUPTIBLE ，然后 schedule\_timeout**()**

wait\_event\_interruptible\_timeout**:**

将当前进程的状态设置为TASK\_INTERRUPTIBLE **,** 然后 schedule\_timeout**()**

TASK\_INTERRUPTIBLE 与 TASK\_UNINTERRUPTIBLE 区别在于，它的休眠是否会被信号打断，别的进程发来一个信号比如kill,TASK\_INTERRUPTIBLE就会醒来去处理。然而 TASK\_UNINTERRUPTIBLE 不会。schedule()，进程调度，而schedule\_timeout()进行调度之后，一定时间后自动唤醒。

对应于不同的进程状态，使用不同的唤醒函数：

void wake\_up**(**wait\_queue\_head\_t **\***queue**);**

void wake\_up\_interruptible**(**wait\_queue\_head\_t **\***queue**);**

唤醒时很有意思，比如你调用 wake\_up 去唤醒一个使用 wait\_event 等，进入休眠的进程，唤醒之后，它会判断 condition 是否为真，如果还是假的继续睡眠。

#### 2.3.2 手动睡眠

方式1：

//建立并初始化一个等待队列项

DEFINE\_WAIT**(**my\_wait**)**

//将等待队列项添加到等待队列头中，并设置进程的状态

prepare\_to\_wait**(**wait\_queue\_head\_t **\***queue**,** wait\_queue\_t **\***wait**,** int state**)**

//进程调度

schedule**()**

//schedule返回，完成后续的清理工作

finish\_wait**()**

方式2：

//建立并初始化一个等待队列项：

DEFINE\_WAIT**(**my\_wait**)**

//将等待队列项添加到等待队列头中：

add\_wait\_queue

//设置进程状态

\_\_set\_current\_status**(**TASK\_INTERRUPTIBLE**);**

//进程调度

schedule**()**

//将等待队列项从等待队列中移除并设置进程状态为运行

remove\_wait\_queue**()**

**set\_current\_state(TASK\_RUNNING)**

其实，这种休眠方式相当于把手工休眠方式一中的第二步prepare\_to\_wait拆成两步做了，即prepare\_to\_wait <====add\_wait\_queue + \_\_set\_current\_status，其他都是一样的。

其实自动睡眠的实现方式，和手动睡眠代码上实现上几乎是一样的。

#define \_\_wait\_event(wq, condition) \

do { \

DEFINE\_WAIT(\_\_wait); \

\

for (;;) { \

prepare\_to\_wait(&wq, &\_\_wait, TASK\_UNINTERRUPTIBLE); \

if (condition) \

break; \

schedule(); \

} \

finish\_wait(&wq, &\_\_wait); \

} while (0)

### 2.4 并发处理

#### 2.4.1 中断屏蔽

local\_irq\_disable**()** //屏蔽中断·

**...**

critical section //临界区

**...**

local\_irq\_enable**()** //开中断

local\_irq\_disable/enable只能禁止/使能本CPU内的中断，不能解决SMP多CPU引发的竞态，故不推荐使用，其适宜于自旋锁联合使用。

#### 2.4.2 原子操作

//1.设置原子变量的值

//输入原子指针，将原子变量置为i

static \_\_inline\_\_ void atomic\_set**(**atomic\_t **\***v**,** int i**);**

//直接将v的原子变量初始化为i

atomic\_t v **=** ATOMIC\_INIT**(**i**);**

//2.原子变量的基本操作

//读值

atomic\_read**(**atmic\_t **\***v**);**

//加/减i操作

void atomic\_add**/**sub**(**int i**,** atomic\_t **\***v**);**

//自加/自减操作

void atomic\_inc**/**dec**(**atomic\_t **\***v**);**

//自加/自减后测试，为0返回ture，否则返回false

int atomic\_inc**/**dec\_test**(**atomic\_t **\***v**);**

//减i后测试，为0返回ture，否则返回false

int atomic\_sub\_and\_test**(**int i**,** atomic\_t **\***v**);**

//加/减i后return

int atomic\_add**/**sub\_return**(**int i**,** atomic\_t **\***v**);**

//自加/自减后return

int atomic\_inc**/**dec\_return**(**atomic\_t **\***v**);**

#### 2.4.3 自旋锁

自旋锁是一种典型的对临界资源进行互斥访问的手段。  
为了获得一个自旋锁，在某CPU上运行的代码需先执行一个原子操作，该操作测试并设置(test-and-set)某个内存变量，由于它是原子操作，所以在该操作完成之前，其他单元无法访问这个内存变量。

//定义自旋锁

spinlock\_t lock**;**

//初始化自旋锁

spin\_lock\_init**(**lock**);**

//获得自旋锁

spin\_lock**(**lock**);** //若获得则返回，否则自旋

tryspin\_lock**(**lock**);** //若获得返回真，否则返回假

//释放自旋锁

spin\_unlock**(**lock**);**

//自旋锁主要针对SMP或单CPU但内核可以抢占的情况，其他系统或不可抢占的CPU中，自旋锁为空操作

驱动程序中应该谨慎使用自旋锁，原因如下：  
1、自旋锁是忙等待锁，当等待时间较长的时候将降低系统系能；  
2、自旋锁可能导致系统死锁（锁陷阱）；  
3、自旋锁锁定器件不能调用可能引起进程调度的函数。如果进程获得自旋锁之后再阻塞，如调用copy\_from\_user()、copy\_to\_user()、kmalloc()和msleep()等函数，则可能导致内核崩溃。

当所要保护的临界访问时间比较短时，用自旋锁是非常方便的，因为它节省了上下文切换的时间。但是CPU得不到自旋锁是，CPU会原地打转，直到其他执行单元解锁为止，所以要求锁不能在临界区里停留时间过长。

自旋锁的用法：

spinlock\_t lock**;**

spin\_lock\_init**(&**lock**);**

spin\_lock**(&**lock**);** //获取自旋锁，保护临界区

。。。。临界区

spin\_unlock**(&**lock**);**//释放自旋锁

#### 2.4.4 读写锁

读写锁允许读的并发。在写操作方面，只能最多有一个写进程，在读操作方面，同时可以有多个读执行单元。当然，读和写也不能同时进行。

// 定义和初始化读写自旋锁

rwlock\_t my\_rwlock **=** RW\_LOCK\_UNLOCKED**;** // 静态初始化

rwlock\_t my\_rwlock**;**

rwlock\_init**(&**my\_rwlock**);** // 动态初始化

// 读锁定：在对共享资源进行读取之前，应先调用读锁定函数，完成之后调用读解锁函数

void read\_lock**(**rwlock\_t **\***lock**);**

void read\_lock\_irqsave**(**rwlock\_t **\***lock**,** unsigned long flags**);**

void read\_lock\_irq**(**rwlock\_t **\***lock**);**

void read\_lock\_bh**(**rwlock\_t **\***lock**);**

// 读解锁

void read\_unlock**(**rwlock\_t **\***lock**);**

void read\_unlock\_irqrestore**(**rwlock\_t **\***lock**,** unsigned long flags**);**

void read\_unlock\_irq**(**rwlock\_t **\***lock**);**

void read\_unlock\_bh**(**rwlock\_t **\***lock**);**

// 写锁定：在对共享资源进行写之前，应先调用写锁定函数，完成之后调用写解锁函数

void write\_lock**(**rwlock\_t **\***lock**);**

void write\_lock\_irqsave**(**rwlock\_t **\***lock**,** unsigned long flags**);**

void write\_lock\_irq**(**rwlock\_t **\***lock**);**

void write\_lock\_bh**(**rwlock\_t **\***lock**);**

int write\_trylock**(**rwlock\_t **\***lock**);**

// 写解锁

void write\_unlock**(**rwlock\_t **\***lock**);**

void write\_unlock\_irqsave**(**rwlock\_t **\***lock**,** unsigned long flags**);**

void write\_unlock\_irq**(**rwlock\_t **\***lock**);**

void write\_unlock\_bh**(**rwlock\_t **\***lock**);**

读写锁的基本用法：

rwlock\_t lock**;** // 定义rwlock

rwlock\_init**(&**lock**);** // 初始化rwlock

// 读时获取锁

read\_lock**(&**lock**);**

**...** // 临界资源

read\_unlock**(&**lock**);**

// 写时获取锁

write\_lock\_irqsave**(&**lock**,** flags**);**

**...** // 临界资源

write\_unlock\_irqrestore**(&**lock**,** flags**);**

#### 2.4.5 顺序锁

顺序锁是对读写锁的一种优化。

1.读执行单元绝对不会被写执行单元阻塞。即读执行单元可以在写执行单元对被顺序锁保护的共享资源进行写操作的同时仍然可以继续读,而不必等待写执行单元完成之后再去读,同样,写执行单元也不必等待所有的读执行单元读完之后才去进行写操作  
2.写执行单元与写执行单元之间仍然是互斥的。  
3.如果读执行单元在读操作期间,写执行单元已经发生了写操作,那么,读执行单元必须重新去读数据,以便确保读到的数据是完整的。  
4.要求共享资源中不能含有指针。

seqlock\_init**(**x**);**       //动态初始化

DEFINE\_SEQLOCK**(**x**);**     //静态初始化

//获得顺序锁

void write\_seqlock**(**seqlock\_t **\***s1**);**  //写加锁

int write\_tryseqlock**(**seqlock\_t **\***s1**);** //尝试写加锁

write\_seqlock\_irqsave**(**lock**,** flags**)**

write\_seqlock\_irq**(**lock**)**

write\_seqlock\_bh**(**lock**)**

//释放顺序锁

void write\_sequnlock**(**seqlock\_t **\***s1**);** //写解锁

write\_sequnlock\_irqrestore**(**lock**,** flags**)**

write\_sequnlock\_irq**(**lock**)**

write\_sequnlock\_bh**(**lock**)**

//读操作:

unsigned int read\_seqbegin**(**const seqlock\_t**\*** sl**);**

read\_seqbegin\_irqsave**(**lock**,** flags**)**

//读执行单元在访问共享资源时要调用顺序锁的读函数,返回顺序锁s1的顺序号;该函数没有任何获得锁和释放锁的开销,只是简单地返回顺序锁当前的序号;

//重读操作:

int read\_seqretry**(**const seqlock\_t**\*** sl**,** unsigned start**);**

read\_seqretry\_irqrestore**(**lock**,** iv**,** flags**);**

//在顺序锁的一次读操作结束之后,调用顺序锁的重读函数,用于检查是否有写执行单元对共享资源进行过写操作;如果有,则需要重新读取共享资源;iv为顺序锁的顺序号;

顺序锁的使用方法：

write\_seqlock**(&**lock**);**

**......** //写操作代码

write\_sequnlock**(&**lock**);**

unsigned int seq\_num **=** 0**;**

**do**

**{**

  seq\_num **=** read\_seqbegin**(&**seqlock**);**

  //读操作代码

**......**

**}** **while(**read\_seqretry**(&**seqlock**,** seq\_num**));**

#### 2.4.6 读-拷贝-更新RCU

对于被RCU保护的功效数据结构，读执行单元不需要获得任何锁就可以访问它，不使用原子指令，而且在除alpha的所有架构上也不需要内存屏障（Memory Barrier），因此不会导致锁竞争、内存延迟以及流水线停滞。使用RCU的写执行单元在访问它前需要首先拷贝一个副本，然后对副本进行修改，最后使用一个回调机制在适当的实际把指向原来数据的指针重新指向新的被修改的数据，这个时机就是所有引用该数据的CPU都退出对共享数据的操作的时候。

RCU可以看作读写锁的高性能版本，相比读写锁，RCU的优点在于既允许多个读执行单元同时访问被保护的数据，又允许多个读执行单元和多个写执行单元同时访问被保护的数据。但是RCU不能替代读写锁，因为如果写比较多时，对读执行单元的性能提高不能弥补写执行单元导致的损失。在绝大部分为读而只有极少部分为写的情况下，它是非常高效的。

RCU主要针对的数据对象是链表，目的是提高遍历读取数据的效率，为了达到目的使用RCU机制读取数据的时候不对链表进行耗时的加锁操作。这样在同一时间可以有多个线程同时读取该链表，并且允许一个线程对链表进行修改（修改的时候，需要加锁）。

在RCU的实现过程中，我们主要解决以下问题：

(1)在读取过程中，另外一个线程删除了一个节点。删除线程可以把这个节点从链表中移除，但它不能直接销毁这个节点，必须等到所有的读取线程读取完成以后，才进行销毁操作。RCU中把这个过程称为宽限期（Grace period）。  
(2)在读取过程中，另外一个线程插入了一个新节点，而读线程读到了这个节点，那么需要保证读到的这个节点是完整的。这里涉及到了发布-订阅机制（Publish-Subscribe Mechanism）。  
(3)保证读取链表的完整性。新增或者删除一个节点，不至于导致遍历一个链表从中间断开。但是RCU并不保证一定能读到新增的节点或者不读到要被删除的节点。

**使用方法：这里没太看明白**

#### 2.4.7 信号量

信号量(semaphore)与自旋锁相同，只有得到信号量才能执行临界区代码，但，当获取不到信号量时，进程不会原地打转而是进入休眠等待状态。

//信号量的结构

struct semaphore sem**;**

//初始化信号量

void sema\_init**(**struct semaphore **\***sem**,** int val**)**

//获取信号量

down**(**struct semaphore **\***sem**)**

down\_trylock**(**struct semaphore **\***sem**)**

down\_interruptible**(**struct semaphore **\***sem**)** //能被信号打断，返回非0

//释放信号量

up**(**struct semaphore **\***sem**)**

#### 2.4.8 互斥锁

//初始化

struct mutex my\_mutex

mutex\_init**(&**my\_mutex**)**

//上锁

mutex\_lock**(**struct mutex **\***lock**)**

mutex\_lock\_interruptible**(**struct mutex **\***lock**)**

mutex\_trylock**(**struct mutex **\***lock**)**

//解锁

mutex\_unlock**(**struct mutex **\***lock**)**

#### 2.4.9 完成量

表示一个执行单元需要等待另一个执行单元完成某事后方可执行。

//1.初始化

struct completion my\_completion**;**    //定义完成量my\_completion

init\_completion**(&**my\_completion**);**    //初始化完成量my\_completion

//2.进入等待

void wait\_for\_completion**(**struct completion**\*** comp**)**

//该函数等待一个完成量被唤醒。该函数会阻塞调用进程,如果所等待的完成量没有被唤醒,那就一直阻塞下去,而且不会被信号打断;

int wait\_for\_completion\_interruptible**(**struct completion**\*** comp**)**

//该函数等待一个完成量被唤醒。但是它可以被外部信号打断;

unsigned long wait\_for\_completion\_timeout**(**struct completion**\*** comp**,** unsigned long timeout**)**

//该函数等待一个完成量被唤醒。该函数会阻塞调用进程,如果所等待的完成量没有被唤醒,调用进程也不会一直阻塞下去,

//而是等待一个指定的超时时间timeout,当超时时间到达时,如果所等待的完成量仍然没有被唤醒,那就返回;

//超时时间timeout以系统的时钟滴答次数jiffies计算

bool try\_wait\_for\_completion**(**struct completion**\*** comp**)**

//该函数尝试等待一个完成量被唤醒。不管所等待的完成量是否被唤醒,该函数都会立即返回

bool completion\_done**(**struct completion**\*** comp**)**

//该函数用于检查是否有执行单元阻塞在完成量comp上(是否已经完成),返回0,表示有执行单元被完成量comp阻塞;

//相当于wait\_for\_completion\_timeout()中的timeout=0

//3.唤醒等待

void complete**(**struct completion**\*** comp**)**

//该函数只唤醒一个正在等待完成量comp的执行单元

void complete\_all**(**struct completion**\*** comp**)**

//该函数唤醒所有正在等待同一个完成量comp的执行单元

#### 2.4.10 对比



### 2.5 select与epoll

设备驱动poll的原型是：

unsigned int **(\***poll**)(**struct file **\***filp**,** struct poll\_table **\***wait**);**

首先，对可能引起设备文件状态变化的等待队列调用poll\_wait(),将对应的等待队列头添加到poll\_table.  
然后，返回表示是否能对设备进行无阻塞读写访问的掩码。

void poll\_wait**(**struct file **\***filp**,** wait\_queue\_head\_t **\***queue**,** poll\_table **\***wait**);**

poll\_wait等效为\_\_pollwait函数。

static void \_\_pollwait**(**struct file **\***filp**,** wait\_queue\_head\_t **\***wait\_address**,**poll\_table **\***p**)**

**{**

struct poll\_table\_entry **\***entry **=** poll\_get\_entry**(**p**);**

**if** **(!**entry**)**

**return;**

get\_file**(**filp**);**

entry**->**filp **=** filp**;**

entry**->**wait\_address **=** wait\_address**;**

init\_waitqueue\_entry**(&**entry**->**wait**,** current**);**

add\_wait\_queue**(**wait\_address**,** **&**entry**->**wait**);**

**}**

将poll进程添加到等待队列。这样，当该队列被唤醒时，处于阻塞的poll也会被唤醒，然后再进行read操作读出想要的值。

基本框架：

unsigned int simple\_poll **(**struct file **\*** filp**,** struct poll\_table\_struct **\*** wait**)**

**{**

unsigned int mask **=** 0**;**

poll\_wait**(**filp**,** **&**simple\_queue\_read**,** wait**);**

poll\_wait**(**filp**,** **&**simple\_queue\_write**,** wait**);**

**if** **(...)** //可读

**{**

mask **|=** POLLIN **|** POLLRDNORM**;**

**}**

**if(...)** //可写

**{**

mask **|=** POLLOUT **|** POLLWRNORM**;**

**}**

**return** mask**;**

**}**

## 3.按键驱动

按键驱动的设计以中断+防抖为主要思路，抖动问题会造成多次中断发生，实则可能是一次按下或释放的操作，这里会使用定时器来去除抖动。

当一次按键按下的时候，可能产生多个脉冲，我们可以等到最后一个脉冲平稳时再真正地做按下或释放的处理。所以，在中断中我们可以不断修改定时器的值，当最后稳定下来，没有中断产生了，就会调用超时函数，再在超时函数里面判断按键状态，返回值给用户空间等。

这里的按键驱动包含了长按 短按 双击的检测，使用到了定时器，poll，睡眠和唤醒等知识。

驱动代码如下：

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/device.h>

#include <linux/io.h>

#include <linux/cdev.h>

#include <linux/irq.h>

#include <linux/delay.h>

#include <linux/workqueue.h>

#include <linux/interrupt.h>

#include <linux/poll.h>

#include <asm/uaccess.h>

#include <asm/arch/map.h>

#include <asm/arch/regs-gpio.h>

#define DEVICE\_NAME "keys"

#define MAX\_KEY\_NUM 4

#define KEY\_DOWN 1//按键按下

#define KEY\_UP 2//按键抬起

#define KEY\_UNCERTAIN 3//按键不确定

struct keys\_dev\_t

**{**

struct cdev cdev**;**

**}**keys\_dev**;**

static volatile int key\_value**;** //确认好的键值

static volatile int key\_status**[**MAX\_KEY\_NUM**];** //记录按键状态

static volatile int key\_count**[**MAX\_KEY\_NUM**];** //记录长按计数

static volatile int key\_last**[**MAX\_KEY\_NUM**];** //双击判断

static struct timer\_list keys\_timer**[**MAX\_KEY\_NUM**];** //按键去抖定时器

static struct timer\_list keys\_double\_timer**[**MAX\_KEY\_NUM**];** //双击判断定时器

static struct class **\***keysdrv\_class**;**

static DECLARE\_WAIT\_QUEUE\_HEAD**(**button\_waitq**);**

/\* 中断事件标志, 中断服务程序将它置1，key\_drv\_read将它清0 \*/

static volatile int ev\_press **=** 0**;**

struct pin\_desc**{**

unsigned int pin**;**

unsigned int irq**;**

char name**[**8**];**

**};**

struct pin\_desc pins\_desc**[**4**]** **=** **{**

**{**S3C2410\_GPF0**,** IRQ\_EINT0**,** "key1"**},**

**{**S3C2410\_GPF2**,** IRQ\_EINT2**,** "key2"**},**

**{**S3C2410\_GPG3**,** IRQ\_EINT11**,**"key3"**},**

**{**S3C2410\_GPG11**,**IRQ\_EINT19**,**"key4"**},**

**};**

struct work\_struct my\_wq**;**/\*定义一个工作队列\*/

void my\_wq\_func**(**unsigned long**);**/\*定义一个处理函数\*/

//中断处理底半部

void my\_wq\_func**(**unsigned long t**)**

**{**

printk**(**"one key preesed:%d\n"**,**key\_value**);**

**}**

static void keys\_double\_timer\_handle**(**unsigned long data**)**

**{**

//等待500ms后，如果定时器还未被删除，则认为是短按

int key **=** data**;**

key\_last**[**key**]** **=** 0**;**

key\_status**[**key**]** **=** KEY\_UP**;**

key\_count**[**key**]** **=** 0**;**

ev\_press **=** 1**;**

schedule\_work**(&**my\_wq**);**//启动后半段

wake\_up\_interruptible**(&**button\_waitq**);** /\* 唤醒休眠的进程 \*/

**}**

static void keys\_timer\_handle**(**unsigned long data**)**

**{**

int key **=** data**;**

int ret**;**

unsigned int pinval**;**

pinval **=** s3c2410\_gpio\_getpin**(**pins\_desc**[**key**].**pin**);**

**if(**pinval**)**

**{**

/\* 松开 \*/

**if(**key\_status**[**key**]** **==** KEY\_UNCERTAIN**)**

**{**

//如果松开了，检测count的值，如果大于50(5秒)，

//则认为是长按

**if(**key\_count**[**key**]** **>** 50**)**

**{**

key\_value **=** key**+**10**;** //长按的键值

key\_status**[**key**]** **=** KEY\_UP**;**

key\_count**[**key**]** **=** 0**;**

ev\_press **=** 1**;**

schedule\_work**(&**my\_wq**);**//启动后半段

wake\_up\_interruptible**(&**button\_waitq**);** //唤醒休眠的进程

**}**

**else**

**{**

//如果松开了，count<50,则认为是短按或者双击

**if(**key\_last**[**key**]** **==** 1**)**

**{**

//如果500ms内又一次被按下

del\_timer**(&**keys\_double\_timer**[**key**]);**

key\_last**[**key**]** **=** 0**;**

key\_value **=** key**+**20**;**

key\_status**[**key**]** **=** KEY\_UP**;**

key\_count**[**key**]** **=** 0**;**

ev\_press **=** 1**;**

schedule\_work**(&**my\_wq**);**//启动后半段

wake\_up\_interruptible**(&**button\_waitq**);**//唤醒休眠的进程

**}**

**else**

**{**

//短按，但是并不能确定接下来的500ms会不会再次按下，形成双击

//所以等待500ms，看看是否会按下。

key\_value **=** key**;**

key\_last**[**key**]** **=** 1**;**

mod\_timer**(&**keys\_double\_timer**[**key**],** jiffies**+**HZ**/**2**);**

**}**

**}**

**}**

**}**

**else**

**{**

/\* 按下 \*/

**if(**key\_status**[**key**]** **==** KEY\_UNCERTAIN**)**

**{**

//确认按键已经按下了

key\_status**[**key**]** **=** KEY\_DOWN**;**

//继续启动定时器，判断是否长按

mod\_timer**(&**keys\_timer**[**key**],** jiffies**+**HZ**/**100**);**

**}**

**else** **if(**key\_status**[**key**]** **==** KEY\_DOWN**)**

**{**

//如果定时器到了，还处于按下的状态，count++

//并继续重启定时器

mod\_timer**(&**keys\_timer**[**key**],** jiffies**+**HZ**/**100**);**

//printk("%d\n",key\_count[key]);

key\_count**[**key**]++;**

**}**

**}**

**}**

/\*

\* 确定按键值

\*/

static irqreturn\_t buttons\_irq**(**int irq**,** void **\***dev\_id**)**

**{**

int key **=** **(**int**)**dev\_id**;**

//设置当前按键的状态为不确定

key\_status**[**key**]** **=** KEY\_UNCERTAIN**;**

//设置当前按键按下去抖定时器的延时并启动定时器

mod\_timer**(&**keys\_timer**[**key**],** jiffies**+**HZ**/**100**);** //设置100ms超时 1HZ=1S

**return** IRQ\_RETVAL**(**IRQ\_HANDLED**);**

**}**

ssize\_t keys\_drv\_read**(**struct file **\***file**,** char \_\_user **\***buf**,** size\_t size**,** loff\_t **\***ppos**)**

**{**

**if** **(**size **!=** 1**)**

**return** **-**EINVAL**;**

//判断非阻塞打开

**if(**file**->**f\_flags **&** O\_NONBLOCK**)**

**{**

**if(**ev\_press **==** 0**)**

**return** **-**EAGAIN**;**

**}**

**else**

**{**

/\* 如果没有按键动作, 休眠 \*/

//ev\_press为0时，休眠

wait\_event\_interruptible**(**button\_waitq**,** ev\_press**);**

**}**

/\* 如果有按键动作, 返回键值 \*/

copy\_to\_user**(**buf**,** **&**key\_value**,** **sizeof(**key\_value**));**

ev\_press **=** 0**;**

**return** 0**;**

**}**

static int keys\_drv\_open**(**struct inode **\***inode**,** struct file **\***file**)**

**{**

/\* 配置GPF0,2为输入引脚 \*/

/\* 配置GPG3,11为输入引脚 \*/

//申请中断

int i**;**

**for(**i **=** 0**;** i **<** MAX\_KEY\_NUM**;** i**++)**

**{**

request\_irq**(**pins\_desc**[**i**].**irq**,** buttons\_irq**,** IRQT\_BOTHEDGE**,** pins\_desc**[**i**].**name**,** **(**void **\*)**i**);**

key\_status**[**i**]** **=** KEY\_UP**;**

key\_count**[**i**]** **=** 0**;**

key\_last**[**i**]** **=** 0**;**

setup\_timer**(&**keys\_timer**[**i**],** keys\_timer\_handle**,** i**);**

setup\_timer**(&**keys\_double\_timer**[**i**],** keys\_double\_timer\_handle**,** i**);**

**}**

**return** 0**;**

**}**

int keys\_drv\_close**(**struct inode **\***inode**,** struct file **\***file**)**

**{**

int i**;**

**for(**i **=** 0**;** i **<** MAX\_KEY\_NUM**;** i**++)**

**{**

free\_irq**(**pins\_desc**[**i**].**irq**,** **(**void **\*)**i**);**

del\_timer**(&**keys\_timer**[**i**]);**

del\_timer**(&**keys\_double\_timer**[**i**]);**

**}**

**return** 0**;**

**}**

static int keys\_drv\_poll**(**struct file **\***file**,** struct poll\_table\_struct **\***wait**)**

**{**

unsigned int mask **=** 0**;**

//添加等待队列到等待队列表中(poll\_table)

poll\_wait**(**file**,** **&**button\_waitq**,** wait**);**

**if(**ev\_press**)**

**{**

//标识数据可以获得

mask **|=** POLLIN **|** POLLRDNORM**;**

**}**

**return** mask**;**

**}**

static struct file\_operations keys\_drv\_fops **=** **{**

**.**owner **=** THIS\_MODULE**,** /\* 这是一个宏，推向编译模块时自动创建的\_\_this\_module变量 \*/

**.**open **=** keys\_drv\_open**,**

**.**read **=** keys\_drv\_read**,**

**.**release **=** keys\_drv\_close**,**

**.**poll **=** keys\_drv\_poll**,**

**};**

int keys\_major**;**

static int keys\_drv\_init**(**void**)**

**{**

dev\_t dev\_id**;**

//1.申请设备号

//注册字符设备编号，在proc/devices中创建keys选项

**if(**keys\_major**)**

**{**

//静态

dev\_id **=** MKDEV**(**keys\_major**,**0**);**

register\_chrdev\_region**(**dev\_id**,**1**,**DEVICE\_NAME**);**

**}**

**else**

**{**

//动态

alloc\_chrdev\_region**(&**dev\_id**,**0**,**1**,**DEVICE\_NAME**);**

keys\_major **=** MAJOR**(**dev\_id**);**

**}**

//2.注册字符设备

//初始化并且注册keys cdev

cdev\_init**(&**keys\_dev**.**cdev**,** **&**keys\_drv\_fops**);**

keys\_dev**.**cdev**.**owner **=** THIS\_MODULE**;**

cdev\_add**(&**keys\_dev**.**cdev**,**dev\_id**,**1**);**

//3.自动创建设备节点

//创建一个类，这个类存放于sysfs下面 /sys/class/keys

keysdrv\_class **=** class\_create**(**THIS\_MODULE**,** DEVICE\_NAME**);**

//mdev daemon就会自动创建节点/dev/keys

//加载模块的时候，用户空间中的mdev会自动响应device\_create(…)函数，去/sysfs下寻找对应的类从而创建设备节点。

class\_device\_create**(**keysdrv\_class**,** **NULL,** MKDEV**(**keys\_major**,** 0**),** **NULL,** DEVICE\_NAME**);**

INIT\_WORK**(&**my\_wq**,(**void **(\*)(**void**\*))**my\_wq\_func**);**/\*初始化工作队列并将其与处理函数绑定\*/

**return** 0**;**

**}**

static void keys\_drv\_exit**(**void**)**

**{**

//1.删除设备节点

//删除/dev/keys设备节点

class\_device\_destroy**(**keysdrv\_class**,**MKDEV**(**keys\_major**,**0**));**

class\_destroy**(**keysdrv\_class**);**

//2.删除字符设备

//删除cdev

cdev\_del**(&**keys\_dev**.**cdev**);**

//3.释放设备号

unregister\_chrdev\_region**(**MKDEV**(**keys\_major**,**0**),**1**);**

**return;**

**}**

module\_init**(**keys\_drv\_init**);**

module\_exit**(**keys\_drv\_exit**);**

MODULE\_LICENSE**(**"GPL"**);**

应用测试代码如下：

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdio.h>

int main**(**int argc**,** char **\*\***argv**)**

**{**

int fd**;**

unsigned int key\_val**;**

fd **=** open**(**"/dev/keys"**,** O\_RDWR**);**

**if** **(**fd **<** 0**)**

**{**

printf**(**"can't open!\n"**);**

**}**

**while** **(**1**)**

**{**

int i**;**

int ret**;**

fd\_set rds**;**

FD\_ZERO**(&**rds**);**

FD\_SET**(**fd**,** **&**rds**);**

ret **=** select**(**fd **+** 1**,** **&**rds**,** **NULL,** **NULL,** **NULL);**

**if(**ret **<** 0**)**

**{**

printf**(**"Read Buttons Device Faild!\n"**);**

exit**(**1**);**

**}**

**if(**ret **==** 0**)**

**{**

printf**(**"Read Buttons Device Timeout!\n"**);**

**}**

**else** **if(**FD\_ISSET**(**fd**,** **&**rds**))**

**{**

read**(**fd**,** **&**key\_val**,** 1**);**

printf**(**"key\_val = %d\n"**,** key\_val**);**

**}**

**}**

close**(**fd**);**

**return** 0**;**

**}**

当按键应用程序运行时(通过后台)，可以通过/proc/interrupts查询发生中断的次数。

# cat /proc/interrupts

CPU0

16**:** 10 s3c**-**ext0 key1

18**:** 4 s3c**-**ext0 key2

30**:** 244717 s3c S3C2410 Timer Tick

32**:** 0 s3c s3c2410**-**lcd

33**:** 0 s3c s3c**-**mci

34**:** 0 s3c I2SSDI

35**:** 0 s3c I2SSDO

37**:** 12 s3c s3c**-**mci

42**:** 0 s3c ohci\_hcd**:**usb1

43**:** 0 s3c s3c2440**-**i2c

51**:** 406 s3c**-**ext eth0

55**:** 4 s3c**-**ext key3

60**:** 0 s3c**-**ext s3c**-**mci

63**:** 18 s3c**-**ext key4

70**:** 1031 s3c**-**uart0 s3c2440**-**uart

71**:** 26244 s3c**-**uart0 s3c2440**-**uart

79**:** 0 s3c**-**adc s3c2410\_action

80**:** 0 s3c**-**adc s3c2410\_action

83**:** 0 **-** s3c2410**-**wdt

Err**:** 0